COMMSYN: On-Chip Communication Architecture Synthesis for Multi-Processor Systems-on-Chip

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Sudeep Pasricha

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Eli Bozorgzadeh
Professor Fadi Kurdahi

2008
The dissertation of Sudeep Pasricha
is approved and is acceptable in quality and form for
publication on microfilm and digital formats:

________________________________
________________________________
________________________________
________________________________
Committee Chair

University of California, Irvine
2008
# Table of Contents

List of Figures ........................................................................................................................ vi
List of Tables ............................................................................................................................ ix
Acknowledgements .................................................................................................................. x
Curriculum Vitae ....................................................................................................................... xii
Abstract of the Dissertation ..................................................................................................... xiv

Chapter 1 Introduction ............................................................................................................ 1
  1.1 On-Chip Communication Architectures in MPSoCs ................................................ 4
      1.1.1 Types of on-chip communication architectures ........................................ 5
      1.1.2 Impact of increasing application complexity ....................................... 8
      1.1.3 Impact of technology scaling ............................................................. 10
  1.2 On-Chip Communication Architecture Design in a Typical MPSoC Design Flow ....... 12
  1.3 Drawbacks of Traditional On-Chip Communication Architecture Synthesis Approach ........................................................................................................ 12
  1.4 Dissertation Contributions ..................................................................................... 15
  1.5 Dissertation Outline .............................................................................................. 18

Chapter 2 CCATB: Fast and Accurate Simulation Models for On-Chip Communication Architecture Exploration ......................................................................... 23
  2.1 Introduction ........................................................................................................... 23
      2.1.1 Chapter Overview ................................................................................ 24
      2.1.2 Related Work ....................................................................................... 25
  2.2 On-Chip Communication Architectures: Background ........................................... 29
      2.2.1 Terminology ......................................................................................... 29
      2.2.2 On-Chip Communication Architecture Standards .................................. 32
  2.3 CCATB Overview .................................................................................................. 35
      2.3.1 Modeling Abstraction .......................................................................... 35
      2.3.2 Modeling Language ............................................................................ 36
      2.3.3 Integrating CCATB into an MPSoC Design Flow .................................. 37
      2.3.4 Component Model Characteristics .................................................... 39
      2.3.5 Maintaining Cycle Count Accuracy with CCATB ............................... 41
  2.4 Simulation Speedup using CCATB ........................................................................ 46
  2.5 Design Space Exploration Case Studies ............................................................... 55
      2.5.1 Case Study 1: Broadband Communication SoC ...................................... 56
      2.5.2 Case Study 2: Multimedia SoC Subsystem .......................................... 61
  2.6 Simulation and Modeling Effort Comparison ......................................................... 67
  2.7 Conclusions ............................................................................................................ 72

Chapter 3 CAPE: Early Power Estimation for On-Chip Communication Architectures using Energy Macro Models ................................................................. 73
3.1 Introduction ........................................................................................................... 73
  3.1.1 Chapter Overview ........................................................................................ 74
  3.1.2 Related Work ............................................................................................... 75
3.2 AMBA AHB Bus Matrix Communication Architecture Overview .............. 76
3.3 Energy Macro-Models ....................................................................................... 78
  3.3.1 Introduction to Macro Modeling .................................................................. 78
  3.3.2 Macro-Model Generation Methodology ...................................................... 79
  3.3.3 Bus Matrix Energy Macro-Models .............................................................. 81
3.4 Experiments ........................................................................................................ 86
  3.4.1 Energy Macro-model Generation ................................................................ 86
  3.4.2 Accuracy of Energy Macro-model ............................................................... 88
3.5 Conclusion .......................................................................................................... 91

Chapter 4 Automated Multi-Constraint On-Chip Communication Architecture
  Synthesis .................................................................................................................. 93
  4.1 Introduction ........................................................................................................ 93
    4.1.1 Chapter Overview ...................................................................................... 94
    4.1.2 Related Work ........................................................................................... 95
  4.2 Full and Partial AMBA AHB Bus Matrix ....................................................... 97
  4.3 Synthesis Framework ....................................................................................... 99
    4.3.1 Background ................................................................................................ 99
    4.3.2 Assumptions and Goals............................................................................. 100
    4.3.3 Simulation Environment .......................................................................... 101
    4.3.4 Communication Parameter Constraint Set .............................................. 102
    4.3.5 Synthesis Flow ........................................................................................ 103
  4.4 Experiments ...................................................................................................... 111
    4.4.1 Quality of Synthesis Framework ............................................................. 112
    4.4.2 Trade-off Generation with Synthesis Framework .................................... 118
  4.5 Conclusion ....................................................................................................... 123

Chapter 5 FABSYN: Floorplan-Aware On-Chip Communication Architecture
  Synthesis ................................................................................................................ 125
  5.1 Introduction ..................................................................................................... 125
    5.1.1 Chapter Overview .................................................................................... 126
    5.1.2 Related Work .......................................................................................... 127
  5.2 Assumptions and Problem Formulation .......................................................... 128
  5.3 Floorplanning and Wire Delay Estimation Engines ........................................... 130
  5.4 Synthesis Framework ...................................................................................... 134
  5.5 Case Studies ................................................................................................... 143
  5.6 Conclusion ..................................................................................................... 157

Chapter 6 COSMECA: On-Chip Memory and Communication Architecture Co-
synthesis ................................................................................................................ 159
  6.1 Introduction .................................................................................................... 159
    6.1.1 Chapter Overview .................................................................................... 161
    6.1.2 Related Work .......................................................................................... 162
  6.2 Memory Subsystem .......................................................................................... 164
  6.3 Assumptions and Problem Formulation ......................................................... 165
6.4 Communication-Memory Parameter Constraint Set $\Psi$ ................................. 166
6.5 COSMECA Co-Synthesis Flow ....................................................................... 167
6.6 Case Studies .................................................................................................... 175
6.7 Conclusion ...................................................................................................... 188

Chapter 7 Conclusions and Future Work ............................................................. 189
7.1 Conclusions .................................................................................................... 189
7.2 Future Directions ........................................................................................... 194

Bibliography ......................................................................................................... 198
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An example of a System-on-chip (SoC) from the multimedia domain</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>MPSoC bus-based on-chip communication architectures</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Increasing performance requirements for emerging applications [3] [4]</td>
<td>8</td>
</tr>
<tr>
<td>1.4</td>
<td>Relative delay comparison of wires versus process technology [5]</td>
<td>10</td>
</tr>
<tr>
<td>1.5</td>
<td>Trend of total interconnect length on a chip [5]</td>
<td>11</td>
</tr>
<tr>
<td>1.6</td>
<td>MPSoC design flow and on-chip communication architecture design phase</td>
<td>14</td>
</tr>
<tr>
<td>1.7</td>
<td>Traditional on-chip communication architecture design approach</td>
<td>15</td>
</tr>
<tr>
<td>1.8</td>
<td>Proposed on-chip communication architecture synthesis framework (COMMSYN)</td>
<td>18</td>
</tr>
<tr>
<td>2.1</td>
<td>Traditional Modeling Abstractions for Communication Space Exploration</td>
<td>26</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of a System with a Bus-based Communication Architecture</td>
<td>30</td>
</tr>
<tr>
<td>2.3</td>
<td>System Design Flow with CCATB</td>
<td>37</td>
</tr>
<tr>
<td>2.4</td>
<td>CCATB Transaction Example</td>
<td>39</td>
</tr>
<tr>
<td>2.5</td>
<td>Reference AMBA 2.0 call sequences</td>
<td>43</td>
</tr>
<tr>
<td>2.6</td>
<td>CCATB transaction execution sequence</td>
<td>44</td>
</tr>
<tr>
<td>2.7</td>
<td>CCATB simulation flow</td>
<td>48</td>
</tr>
<tr>
<td>2.8</td>
<td>GatherRequests procedure</td>
<td>49</td>
</tr>
<tr>
<td>2.9</td>
<td>HandleBusRequests procedure</td>
<td>50</td>
</tr>
<tr>
<td>2.10</td>
<td>HandleCompletesRequests procedure</td>
<td>50</td>
</tr>
<tr>
<td>2.11</td>
<td>ArbitrateRequest procedure</td>
<td>51</td>
</tr>
<tr>
<td>2.12</td>
<td>UpdateDelaysAndSets procedure</td>
<td>52</td>
</tr>
<tr>
<td>2.13</td>
<td>DetermineIncrementPeriod procedure</td>
<td>53</td>
</tr>
<tr>
<td>2.14</td>
<td>IncrementSimulationTime procedure</td>
<td>54</td>
</tr>
<tr>
<td>2.15</td>
<td>Broadband Communication SoC platform</td>
<td>56</td>
</tr>
<tr>
<td>2.16</td>
<td>Bus protocol comparison</td>
<td>57</td>
</tr>
<tr>
<td>2.17</td>
<td>Arbitration strategy comparison</td>
<td>57</td>
</tr>
<tr>
<td>2.18</td>
<td>Topology configuration A</td>
<td>59</td>
</tr>
<tr>
<td>2.19</td>
<td>Topology configuration B</td>
<td>59</td>
</tr>
<tr>
<td>2.20</td>
<td>Topology configuration comparison</td>
<td>61</td>
</tr>
<tr>
<td>2.21</td>
<td>Varying SDRAM OO Queue size</td>
<td>61</td>
</tr>
<tr>
<td>2.22</td>
<td>SoC Multimedia Subsystem</td>
<td>62</td>
</tr>
<tr>
<td>2.23</td>
<td>SoC Communication Architecture Topologies</td>
<td>63</td>
</tr>
<tr>
<td>2.24</td>
<td>SoC platform</td>
<td>68</td>
</tr>
<tr>
<td>2.25</td>
<td>Simulation Speed Comparison</td>
<td>68</td>
</tr>
<tr>
<td>3.1</td>
<td>AMBA AHB Full Bus Matrix</td>
<td>77</td>
</tr>
<tr>
<td>3.2</td>
<td>Energy Macro-model Generation Methodology</td>
<td>79</td>
</tr>
<tr>
<td>3.3</td>
<td>Macro-model Template</td>
<td>80</td>
</tr>
<tr>
<td>3.4</td>
<td>Average cycle energy estimation errors</td>
<td>88</td>
</tr>
<tr>
<td>3.5</td>
<td>Predicted and measured power waveforms</td>
<td>90</td>
</tr>
<tr>
<td>4.1</td>
<td>AMBA AHB Full Bus Matrix</td>
<td>97</td>
</tr>
</tbody>
</table>
Figure 4.2 AMBA AHB Partial Bus Matrix ................................................................. 98
Figure 4.3 Communication Throughput Graph (CTG).............................................. 100
Figure 4.4 Bus Matrix Synthesis Framework Flow ................................................... 104
Figure 4.5 bound function ........................................................................................ 108
Figure 4.6 Branch and bound clustering illustration ................................................ 109
Figure 4.7 CTG for SIRIUS MPSoC application ....................................................... 113
Figure 4.8 Synthesized bus matrix for SIRIUS MPSoC .......................................... 115
Figure 4.9 Comparison with threshold based approach for SIRIUS ......................... 116
Figure 4.10 Comparison of number of buses for MPSoC applications ..................... 117
Figure 4.11 CTG for networking MPSoC application .............................................. 119
Figure 4.12 Power-Performance-Cost Trade-off graph .......................................... 120
Figure 4.13 Component-wise Power Comparison ..................................................... 120
Figure 4.14 Total Energy vs. Runtime Trade-off graph .......................................... 121
Figure 4.15 Power vs. Chip-Area Trade-off graph for MPSoC ................................. 122
Figure 5.1 Communication Throughput Graph (CTG) ............................................ 129
Figure 5.2 Transforming multiple pin net into a two pin net .................................... 131
Figure 5.3 Example floorplan .................................................................................... 133
Figure 5.4 Hierarchical bus architecture synthesis flow ......................................... 135
Figure 5.5 preprocess procedure ............................................................................ 136
Figure 5.6 mutate_topology procedure ................................................................. 140
Figure 5.7 optimize design procedure ..................................................................... 141
Figure 5.8 Network Communication SoC Subsystem.............................................. 144
Figure 5.10 Synthesized SoC subsystem .................................................................... 145
Figure 5.11 Floorplan for SoC subsystem ................................................................. 145
Figure 5.12 Network Communication SoC Subsystem Derivative ......................... 147
Figure 5.13 Synthesized SoC subsystem for derivative architecture ....................... 148
Figure 5.14 Final floorplan for derivative SoC subsystem ........................................ 148
Figure 5.15 MPSoC networking subsystem ............................................................... 150
Figure 5.16 Intermediate configuration for MPSoC networking subsystem ............. 152
Figure 5.17 Floorplan for intermediate configuration ............................................... 153
Figure 5.18 Final synthesized configuration for MPSoC networking subsystem ....... 153
Figure 5.19 Final floorplan for synthesized MPSoC networking subsystem ............. 154
Figure 6.1 Comparison of memory and communication architecture synthesis approaches for (a) MPSoC example, (b) result of performing memory synthesis before communication architecture synthesis, and (c) result of performing co-synthesis of memory and communication architectures ..................................................... 160
Figure 6.2 COSMECA co-synthesis flow ................................................................. 168
Figure 6.3 COSMECA co-synthesis example ........................................................... 170
Figure 6.4 memmap heuristic ................................................................................... 173
Figure 6.5 PYTHON Communication Throughput Graph (CTG) ............................. 177
Figure 6.6 Synthesized output for PYTHON ............................................................ 179
Figure 6.7 PYTHON final solution space (for N=10) .............................................. 179
Figure 6.8 Effect of varying threshold value on solution quality for PYTHON .......... 180
Figure 6.9 SIRIUS Communication Throughput Graph (CTG) ............................... 181
Figure 6.10 Synthesized output for SIRIUS ............................................................. 183
Figure 6.11 SIRIUS final solution space (for N=10) .......................................................... 184
Figure 6.12 Effect of varying threshold value on solution quality for SIRIUS .......... 185
Figure 6.13 Comparison of best solution bus count ....................................................... 187
Figure 6.14 Comparison of best solution memory area ................................................. 187
List of Tables

Table 2.1 Fields in token structure ................................................................. 40
Table 2.2 Execution cycle counts (in millions of cycles) ................................... 64
Table 2.3 Comparison of speedup and modeling effort .................................... 69
Table 2.4 Comparison of simulation time during exploration .......................... 71
Table 3.1 Coefficients for Energy Macro-Models (180nm) ............................... 85
Table 3.2 Coefficients for Input Stage Bus Component (in pJ) ......................... 86
Table 3.3 Comparing time taken for power estimation ..................................... 90
Table 4.1 Dependent Variable Activations for a Write Transaction ................. 111
Table 4.2 Throughput Constraint Paths (TCPs) For SIRIUS ......................... 114
Table 4.3 Customizable Parameter Constraint Set (SIRIUS) .......................... 114
Table 4.4 Number of Cores in MPSOC Applications ..................................... 116
Table 4.5 Throughput Constraint Paths (TCPs) ............................................ 119
Table 5.1 Parameters based on NTRS 97 ..................................................... 133
Table 5.2 Customizable Parameter Set ....................................................... 144
Table 5.3 Communication Parameter Values .............................................. 146
Table 5.4 Customizable Parameter Set ....................................................... 147
Table 5.5 Communication Parameter Values (Derivative Arch.) ..................... 149
Table 5.6 Throughput Constraint Paths (TCPs) ............................................ 150
Table 5.7 Customizable Parameter Set ....................................................... 151
Table 5.8 Communication Parameter Values .............................................. 154
Table 5.9 Synthesis Result Comparison ...................................................... 155
Table 6.1 Core Distribution in MPSOC Applications ...................................... 176
Table 6.2 PYTHON Throughput Constraint Paths (TCPs) ............................. 177
Table 6.3 PYTHON Global Constraint Set $\Psi_G$ .......................................... 178
Table 6.4 SIRIUS Throughput Constraint Paths (TCPs) ............................... 182
Table 6.5 SIRIUS Global Constraint Set $\Psi_G$ ........................................... 183
Table 6.6 Co-Synthesis Time for MPSOC Applications ................................. 186
Acknowledgements

First and foremost, I would like to thank my advisor, Prof. Nikil Dutt for his invaluable support and guidance during my graduate studies. Without his excellent insights, advice and encouragement, this dissertation would not exist. He gave me the freedom I needed to pursue my research interests, while at the same time ensuring that I did not lose sight of the “big picture”. I am indebted to him for the time, patience, and effort he has spent in my education. I am grateful to Prof. Elaheh Bozorgzadeh and Prof. Fadi Kurdahi for their tireless encouragement and guidance during our numerous brainstorming sessions related to my dissertation area over the past four years. I would like to acknowledge Prof. Alex Veidenbaum and Prof. Nalini Subramanian for their very interesting graduate courses which inspired me to do research that turned into my very first research publications in prestigious international conferences, and later, journals. I would also like to thank Prof. Tony Givargis for his valuable comments and input on this research.

This dissertation was to a large extent inspired by my work at Conexant Inc. I would especially like to thank Dr. Mohamed Ben-Romdhane for laying the foundations of the problems that I would go on to tackle in this dissertation. His incredible support, vision, and encouragement when I was an intern at Conexant enabled me to understand the problems facing designers and make the most of my industrial experience. I am also grateful to Dhanumjai Pasumarthy for tutoring me on logic synthesis tool flows, and Vincent Chang for our many discussions on the AMBA interface standard at Conexant.
I am thankful to my many friends and colleagues from CECS and the ACES lab who helped make this work possible. I am not only grateful for the numerous interesting and useful technical discussions that we have had, but also for their friendship, advice and support. It is, quite simply, hard to find nicer people to work with and learn from. I would especially like to acknowledge Prabhat Mishra, Young-Hwan Park, Gabor Madl, Qiang Zhu, Luis Baten, Shivajit Mohapatra, Aseem Gupta, Partha Biswas, Aviral Shrivastava, Houman Homayoun, Peter Grun, Nick Savoiu, and Manev Luthra, who have, directly or indirectly, collaborated on my research. I would also like to thank my many other friends at UCI, Ilya Issenin, Sudarshan Banerjee, Kyoungwoo Lee, Minyoung Kim, Andreas Gerstlauer, Shireesh Verma, Samar Abdi, Srikant Srinivasan, Mahesh Mamidipaka, Love Singhal, Carmen Badea, Radu Cornea, Jelena Trajkovich, Jelena Ciracovich, Jayram Moornikara, Jeff Furlong, Arup Chakroborty, Mohammad Ali Ghodrat, Amir Hossein Gholami Pour, Arijit Ghosh, Shahin Golshan, Lukai Cai, Dongwan Shin, Jerry Peng, Mehrdad Reshadi, Bita Gorjiara, Gunar Schirner, Pramod Chandraiah, Kiran Ramineni, Fredrick Doucet, and others whose companionship and support has been vital in making my stay so memorable.

I especially thank Melanie Sanders and Lisa Schilling for making CECS such an enjoyable place to work and for all their help and advice on so many different subjects.

Last but by no means the least, I am extremely grateful to my parents and my girlfriend Laurie Rose for their love and continued support throughout my studies.
## Curriculum Vitae

Sudeep Pasricha

### Education

<table>
<thead>
<tr>
<th>Year</th>
<th>Degree</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>Ph.D. in Information and Computer Science</td>
<td>University of California, Irvine</td>
</tr>
<tr>
<td>2005</td>
<td>M.S. in Information and Computer Science</td>
<td>University of California, Irvine</td>
</tr>
<tr>
<td>2000</td>
<td>B.E. in Electronics and Communications Engineering</td>
<td>Delhi Institute of Technology, Delhi, India</td>
</tr>
</tbody>
</table>

### Research and Work Experience

<table>
<thead>
<tr>
<th>Year</th>
<th>Title</th>
<th>Company/Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002 - 2008</td>
<td>Graduate Researcher</td>
<td>University of California, Irvine</td>
</tr>
<tr>
<td>2003 - 2004</td>
<td>Research Intern</td>
<td>Conexant, Newport Beach, CA, USA</td>
</tr>
<tr>
<td>2000 - 2002</td>
<td>Design Engineer</td>
<td>STMicroelectronics, Noida, India</td>
</tr>
</tbody>
</table>

### Selected Publications


S. Pasricha and N. Dutt, "COSMECA: Application Specific Co-Synthesis of Memory and Communication Architectures for MPSoC", IEEE Design Automation and Test in Europe Conference (DATE 2006), Munich, Germany, March 2006.


Abstract of the Dissertation

COMMSYN: On-Chip Communication Architecture Synthesis for Multi-Processor Systems-on-Chip

by

Sudeep Pasricha

Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 2008
Professor Nikil Dutt, Chair

With the increasing performance demands of emerging convergence applications and advances in process technology, Multiprocessor Systems-on-Chip (MPSoCs) are becoming prevalent in modern embedded systems. The on-chip communication architecture that interconnects the various components in such systems has a large and critical impact on the overall system performance, power, cost and reliability. Consequently, the design, exploration and implementation of on-chip communication architectures have become some of the most time-consuming activities for system designers in a typical MPSoC design flow. The task of on-chip communication architecture design is further hampered by a lack of state-of-the-art system-level tools and methodologies to cope with their increasingly significant role in systems today.

In this dissertation, we propose the COMMSYN framework for automated exploration and synthesis of on-chip communication architectures in emerging heterogeneous MPSoC applications. We have developed fast and accurate simulation and power estimation models for reliable on-chip communication architecture exploration early in the design flow. These models are used in our framework to make informed decisions
during synthesis. COMMSYN enables a physical-implementation-aware and memory-architecture-aware on-chip communication architecture synthesis, comprehensively and automatically generating both the topology and protocol parameters, while trading off multiple design constraints such as power, performance, area and cost.

Such a multi-faceted synthesis framework accrues many benefits for MPSoC designs such as improved design reliability and quality, better complexity management, reduced system cost and a faster time-to-market. The experiments on several industrial strength applications demonstrate the utility of the automated and comprehensive synthesis framework for MPSoC designs.
Chapter 1
Introduction

For over four decades, advances in semiconductor technology have led to the evolution of digital computer systems from large, bulky mainframes to small and lightweight devices that can be found in a variety of electronic systems, such as those found in cars, cell phones, digital cameras, microwave ovens, fax machines and card readers. These special purpose computing devices that are embedded in larger electronic systems are commonly referred to as embedded systems. Although they cost far less than desktop computers, embedded systems easily outnumber desktops. While an average American household today might have one or two desktop computers, they have hundreds of embedded computing systems around the house. Billions of these embedded systems are sold every year, compared to several million desktop computing units [1].

It was Intel’s co-founder Gordon Moore who predicted back in the 1960’s that transistor density in integrated circuits will double roughly every 18 months. This prediction has withstood the test of time, and over the last decade has enabled embedded systems that were earlier implemented at the board level, to be shrunk down and implemented on a single chip. These single chip integrated circuits are commonly referred to as Systems-on-Chip (SoC), and typically consists of several complex heterogeneous components such as one or more programmable processors, dedicated hardware to perform specific tasks, on-chip memories, sensors that interact with the outside world and an on-chip communication architecture that interconnects these components together. Figure 1.1 shows an example of one such System-on-Chip from the
multimedia domain, which is made up two ARM9 microprocessors running embedded software, several on-chip memories, DMA and LCD controllers, peripherals (e.g., timer and interrupt controller), and external interfaces (e.g., USB and Ethernet), all of which are integrated via a bus architecture consisting of multiple shared interconnected buses. Such SoC designs typically have tens to hundreds of million transistors integrated on a single chip and are characterized by multi-dimensional constraints. Not only must these designs have high performance, and high reliability, but at the same time must have low power (or energy), low cost and fast time-to-market. These multi-dimensional constraints make the task of designing such SoC systems immensely challenging. Often, design optimizations that enable a particular constraint to be satisfied (e.g., performance) end up violating another constraint (e.g., energy budget).

Figure 1.1 An example of a System-on-chip (SoC) from the multimedia domain
In the last couple of years, these SoC designs have been transforming rapidly into incredibly complex systems. This is in part due to steady technological advances, but to a large extent as a result of the recent trend of convergence in digital system functionality. Consumers today, for instance, are demanding cell phones that can take digital pictures and record video clips, play 3D games and MP3 songs, and have GPS (global positioning system) and PDA (personal digital assistant) capabilities in addition to making phone calls. Consequently, Systems-on-Chip have evolved into Multiprocessor Systems-on-Chip (MPSoC) that have billions of transistors integrated on a chip. These emerging MPSoC designs typically consist of multiple microprocessors (general purpose embedded processors as well as more customized processors such as DSPs), and tens to hundreds of additional components. A popular example of an MPSoC is the IBM CELL [2] which has been used in the Sony Playstation 3 gaming console. It consists of nine processors – eight special-purpose synergistic processing elements (SPE) that perform dedicated computing tasks, and a single general purpose processing unit (GPU) that performs generalized processing, and oversees the activities on the chip. Additionally, the CELL has on-chip L2 memory, interface components to interact with external electronic systems and a ring bus-based on-chip communication architecture that facilitates data communication between the components on the chip.

The on-chip communication architecture fabric in MPSoCs has a significant impact on its power, performance, cost, reliability and time-to-market. This chapter introduces the various aspects of on-chip communication in MPSoCs, and gives insights into why on-chip communication architectures are becoming a critical issue in MPSoC designs. We first introduce the different types of on-chip communication architectures used in MPSoC
designs, and discuss trends in application complexity and technology scaling that makes the task of on-chip communication architecture design so challenging. Next, we illustrate where communication architecture design fits into a typical MPSoC design flow. Subsequently, we describe the drawbacks of current practices in on-chip communication architecture design, and then introduce our proposed approach for efficient and comprehensive on-chip communication architecture synthesis to overcome these drawbacks. Finally, we conclude with an overview of the contributions this dissertation makes and an outline of the remaining chapters.

1.1 On-Chip Communication Architectures in MPSoCs

The components in an MPSoC design invariably need to communicate with each other during application execution. For instance, a microprocessor fetches instructions from memory components, or writes to external memories by sending data to an on-chip memory controller. It is the responsibility of the on-chip communication architecture to ensure that the multiple, co-existing data streams on the chip are correctly and reliably routed from the source components to their intended destinations. In addition to correctness, the on-chip communication architecture must provide latency or bandwidth guarantees to ensure that the application performance constraints are satisfied. A latency guarantee implies that a data unit must traverse the communication architecture and reach its destination within a finite amount of time, determined by a latency bound (e.g., 40 ns from source to destination). A bandwidth guarantee implies that a group of data units must traverse a portion of the communication architecture at a certain data rate, as determined by the bandwidth requirements (e.g., 100 megabits/sec from source to
destination). Depending on the performance requirements of an application, various types of on-chip communication architectures can be used, as described in following subsection.

![Diagram of different on-chip communication architectures]

Figure 1.2 MPSoC bus-based on-chip communication architectures

1.1.1 Types of on-chip communication architectures

The building block of most of the on-chip communication architectures used in MPSoC designs today is the single shared bus. This is the simplest on-chip
communication architecture, consisting of a set of shared, parallel wires to which various components are connected. Only one component on the bus can have control of the shared wires at any given time to perform data transfers. This limits the parallelism and achievable performance in the system, which makes it unsuitable for most MPSoC applications that can have tens to hundreds of components. Consequently, the single shared bus architecture is not scalable to meet the demands of MPSoC applications.

Figure 1.2 shows the various kinds of on-chip communication architectures that are currently being used in MPSoC designs. Today’s MPSoC designs mostly use shared bus-based communication architectures. Figure 1.2 (a) shows a hierarchical shared bus architecture that consists of a hierarchy of buses interconnected using bridge components. Shared buses higher up in the hierarchy are typically operated at higher clock frequencies, and are used to connect high speed, high performance components. On the other hand, shared buses lower down in the hierarchy are operated at lower frequencies to save power, and connect high latency, low performance components. Figure 1.2 (b) shows a ring type bus, similar to that used in the IBM Cell MPSoC. The ring bus is actually a set of unidirectional, concentric and pipelined buses which allow high frequency operation and high bandwidth transfers between components on the bus. Figure 1.2 (c) shows an ad-hoc bus architecture, where buses are operated at different frequencies and components can have point to point links with each other, as needed. Finally, figure 1.2 (d) shows the bus matrix (or crossbar bus) where a crossbar type architecture connects processors (and their local bus components) on the left to memories and peripherals on the right. This kind of architecture is a combination of shared bus and point-to-point interconnections.
Each of the above bus-based on-chip communication architectures is defined by its two major constituents: topology and protocol parameters. The topology of an on-chip communication architecture refers to how the buses are interconnected together, and how the various components are mapped to each of the buses. The protocol parameters refer to such parameters as arbitration schemes, bus widths, bus clock frequencies, buffer sizes and burst transfer sizes, which are specific to the protocol used by the communication architecture. Designing an on-chip communication architecture thus implies determining both its topology and protocol parameter values.

It has been projected that future MPSoC designs with hundreds of components will make use of network-on-chip (NoC) communication fabrics, where instead of shared buses, packet switched network fabrics with routers are used to transfer data between on-chip components. However, NoCs are still in their early phase of conception, and despite a lot of research, few concrete implementations of NoC based MPSoCs exist today. NoCs are also plagued by high power consumption, greater design space complexity and high area overheads, which may hinder their adoption for future MPSoC designs. The focus of this dissertation is on bus-based on-chip communication architectures that are currently the norm for on-chip communication in MPSoCs. It is unlikely that bus-based communication architectures will be phased out anytime soon in the future, because of their low area overhead, extensive tool support and lower power consumption. Even if NoCs are adopted as communication fabrics for MPSoCs in the future, they will encompass several sub-systems that are interconnected using buses. Hence the problem of designing bus-based communication architectures will be just as relevant in the coming years.
Having introduced the different types of on-chip communication architectures used in MPSoC designs, we will now look at how increasing application complexity and technology scaling have affected on-chip communication in recent years.

![Figure 1.3 Increasing performance requirements for emerging applications](image)

**Performance requirements in GOPS**

**Figure 1.3 Increasing performance requirements for emerging applications [3] [4]**

### 1.1.2 Impact of increasing application complexity

With increasing application complexity and the rising number and variety of components being integrated into MPSoC designs, communication between on-chip components is playing an increasingly critical role in ensuring that application performance constraints are satisfied. Due to the increasing inter-dependence of various components on a chip, a seemingly insignificant bottleneck in transferring a single data stream between two components can stall the entire chip, leading to chip failure. This not
so uncommon scenario is an artifact of the sheer number of simultaneous data streams traversing a typical MPSoC chip at any given time, and being managed by finite communication resources (wires, buffers). Figure 1.3 shows the rising performance requirements of emerging applications, which will inevitably increase the amount of data communication traffic on a chip and further increase the probability of unforeseen bottlenecks encountered in on-chip communication in the future.

To cope with the increasing MPSoC performance requirements, on-chip communication architectures have also undergone an evolution in complexity – from shared buses, to hierarchical shared buses, and on to bus matrix (or crossbar bus) architectures. This has had two notable consequences for on-chip communication architecture design. Firstly, since advanced bus-based architectures such as the bus matrix make use of many more wires and logic components to support high performance requirements, they have a much larger power consumption and area overhead. Thus design decisions made during communication architecture selection and implementation must take into account not only the supported performance, but also ensure that overall chip power and area constraints are not violated by the communication architecture. Secondly, these advanced communication architectures have enormous design spaces which are not so easy to explore. The combination of different topologies, component mapping choices, and protocol parameter values for a communication architecture can easily create a design space with billions of possible configurations, making it incredibly difficult for designers to choose which configurations to explore in the finite amount of time available in an MPSoC design cycle. Thus the task of designing on-chip communication architectures today has become a major challenge for designers, requiring
a careful, time-consuming decision process to adequately balance the different constraints (such as power, performance, area) of the application.

![Relative delay comparison of wires versus process technology](image)

**Figure 1.4 Relative delay comparison of wires versus process technology [5]**

### 1.1.3 Impact of technology scaling

The increasing levels of component integration would not have been possible without technology scaling that has enabled a reduction in transistor size, allowing more of them to be integrated into the same area with each passing technology generation. However, these technological advances that have ushered the industry into the deep submicron (DSM) era, with the ongoing commercialization of the 65 nm and 45 nm processes, have introduced new challenges for on-chip communication architecture design. Precise control of the fabrication process in DSM technologies is almost impossible, leading to
process uncertainties that cause non-uniformity of sheet resistance and an increase in coupling noise between adjacent wires in buses. In addition, decreasing wire pitch and increasing aspect ratio with technological advances further accelerates these issues. The end result of these factors is that signal propagation delay on wires (i.e., interconnect delay) is increasing with each technology generation, which puts a limit on the communication performance.

According to ITRS 2005 predictions (Figure 1.4) [5], the gap between interconnection delay and gate delay will increase to 9:1 at the 65 nm technology. This is in sharp contrast to the 2:1 gap between interconnection delay and gate delay at the 180 nm technology. This indicates that communication, and not computation, will be the key performance bottleneck in DSM technologies. In addition, total wire length on a chip is expected to amount to 2.22 km/cm² by the year 2010 (Figure 1.5) [5]. Another
observation is the increase of power dissipation due to the charging and discharging of interconnection wires on a chip. According to Chandrakasan and Brodersen [6], the capacitance portion of the interconnect contributes to about 30% of the total capacitance of a chip, and soon the interconnect will consume about 50 times more power than logic circuits [7]. This means that for MPSoC designs in the DSM era, the performance, power consumption, cost and area will be much more influenced by the on-chip communication architecture than the gates on the chip.

1.2 On-Chip Communication Architecture Design in a Typical MPSoC Design Flow

Before we analyze the current state-of-the-art practices in communication architecture design, it will be useful to understand where on-chip communication architecture design fits into a typical MPSoC design flow. Figure 1.6 shows a typical MPSoC design flow. The starting point is when specifications are received from the customer (or the marketing department) describing the application (e.g., cell phone, MP3 player) that needs to be designed. In the first step, designers select the algorithms to use, perform optimization and create a functional model of the application in a high level language such as C/C++. This is a very high level sequential-execution model of the application which captures its entire functionality. In the next step, designers perform hardware/software partitioning – mapping part of the functionality into hardware components and the remaining functionality to software. Architecture exploration is performed to select the best possible hardware or software candidates for the functional (or behavior) mapping. The result is an architecture model of the system where the
computation entities in the system have been defined. Note however that these entities communicate with each other in an abstract manner, for e.g., using high level message passing. It is the responsibility of the subsequent on-chip communication architecture synthesis step to define a communication architecture for the system. This synthesis step performs design space exploration to select a particular on-chip communication architecture (e.g., hierarchical bus, bus matrix) that best satisfies the constraints of the application. Once a communication architecture has been selected, further exploration is performed to define its topology and values for protocol parameters (e.g., arbitration scheme, bus width and frequency). The resulting model is called a communication model. It is an enhanced architecture model of the system with well defined computation and communication entities. The model is further refined in the next step, in which behavior inside computation blocks is scheduled at cycle boundaries (cycle-accurate behavior). The interface between the computation blocks and the communication architecture is also refined to a pin-accurate level, with all the signals needed for communication being explicitly modeled. These steps lead to the creation of a detailed implementation model. This model is essentially an RTL (register transfer logic) level model, which can be an input to standard logic synthesis tools such as Synopsys Design Compiler or Cadence PKS to create a gate-level netlist of the design. Subsequently, the designer performs placement of the various modules on the chip floorplan, followed by a routing step to connect the modules together. The resulting GDSII file is then sent off to a semiconductor foundry for fabricating the MPSoC design.
This dissertation is concerned with the on-chip communication architecture synthesis phase in the MPSoC design flow, which is shown highlighted in Figure 1.6. Note that this MPSoC flow is a high level illustration of an actual flow, which does not show many of the iteration and verification steps that must be performed simultaneously with the design flow. In addition, while the MPSoC design flow consists of the basic steps as shown in Figure 1.6, in practice designers may sometimes merge a few of these steps, or split one or more of the steps for a more detailed treatment of the problem.
1.3 Drawbacks of Traditional On-Chip Communication Architecture Synthesis Approach

Let us now take a closer look at how on-chip communication architecture synthesis is performed today in industry. Figure 1.7 shows the portion of the MPSoC design flow from Figure 1.6 that is concerned with communication architecture design. There are several critical drawbacks in the approach used for on-chip communication architecture synthesis today. These are described below.

![Diagram of Traditional on-chip communication architecture design approach]

(i) Manual ad-hoc exploration: Designers today perform exploration of the on-chip communication architecture design space manually, by repeatedly running simulations and making performance estimations in an ad-hoc manner. The reason for using this approach is the lack of automated design exploration tools available today, but more importantly because designers are accustomed to manually exploring systems over the
course of the past several years. While a manual, ad-hoc exploration approach guided by designer intuition was feasible and yielded near optimal results for smaller SoC designs with a few components and a small design space in the past, this approach does not scale well for emerging MPSoC designs that have tens to hundreds of components, with a design space that is orders of magnitude larger and more complex than that of smaller SoC systems. Manual exploration of today’s MPSoC designs during on-chip communication architecture exploration leads to inferior designs because designers can only explore a few selected configurations in a reasonable amount of time, with no guarantee that the configuration explored are anywhere near optimal.

(ii) Ineffective simulation models: The simulation models used by designers today to perform exploration of the on-chip communication architecture design space are either too slow, or fast but not accurate enough to offer reliable estimates needed to make crucial design decisions. Because of the slow simulation speed of accurate simulation models, designers can only explore very few configurations. The problem is getting worse for emerging large MPSoC designs that are not only time consuming to simulate, but also take a lot of time to model because of their complexity.

(iii) Lack of early power estimation: The power dissipation of an on-chip communication architecture is increasingly becoming a first class design objective in addition to its performance capabilities, especially for mobile handheld devices that run on batteries which have a fixed energy budget. It is imperative for designers today to make use of the power dissipation information of communication architecture configurations while making design decisions early in the design flow to ensure that the power/energy budget constraints are respected. However, power dissipation information
can only be reliably obtained much later in the design flow, after logic synthesis, once the
gate level netlist has been created. In the absence of any tool support for early power
estimation, designers today do not perform power-aware exploration, or trade-offs
between power and performance when exploring on-chip communication architecture
configurations. Consequently, it is impossible to avoid unforeseen power constraint
violations later in the flow today, which requires costly redesign iterations.

(iv) Lack of physical implementation awareness: In the DSM era, the performance
(and power) of the on-chip communication architecture depends strongly on physical
level information (from much later in the design flow) such as the floorplan and layout of
the components on the chip, which determines the lengths and proximity of bus wires to
each other. However, designers performing on-chip communication architecture synthesis
today do not consider physical level information during synthesis. As a result, the
estimates during early exploration are not accurate, leading to misinformed decisions and
inferior designs. In addition, design decisions made during on-chip communication
architecture synthesis can be practically unrealistic (e.g., selecting very high bus clock
frequencies which are not practically realizable) due to the disconnect with the physical
level, causing severe timing violations that are detected at the physical level and often
result in time consuming redesign iterations.

(v) Lack of memory architecture awareness: The memory architecture is responsible
for a significant amount of traffic on the communication buses, because memories are
heavily used in most applications. Consequently, the design of the memory architecture
has an influence on the effectiveness of different on-chip communication architecture
configurations. Unfortunately, designers today perform memory architecture synthesis
separately from on-chip communication architecture synthesis, and fail to exploit the inherent interdependence between them, leading to sub-optimal designs.

1.4 Dissertation Contributions

In order to address the drawbacks in the current on-chip communication architecture synthesis approaches, in this dissertation we propose the COMMSYN framework for efficient, automated and comprehensive bus-based on-chip communication architecture synthesis. Figure 1.8 shows a high level overview of the novel COMMSYN synthesis framework that overcomes the various shortcomings found in the current synthesis approach (elaborated in the previous section). We describe the specific contributions made in the dissertation below.

![Figure 1.8 Proposed on-chip communication architecture synthesis framework (COMMSYN)](image-url)
(i) Automated Multi-Constraint Topology and Protocol Synthesis: In order to speed up the time-consuming communication architecture synthesis process, we have developed an automated synthesis framework that quickly and effectively traverses regions of interest in the design space. The automated framework makes use of various optimization algorithms and heuristics to prune the practically infeasible regions of the design space, and is able to explore more useful and a larger number of configurations compared to the manual approach, in the same amount of time. While there have been a few research efforts to automate the communication architecture synthesis process, the approaches have either focused on topology synthesis, or protocol parameter synthesis for a fixed topology, but not both. In addition to being automated, our framework is comprehensive – it simultaneously synthesizes the topology, component mappings, and values for a variety of protocol parameters such as arbitration schemes, bus widths, bus clock frequencies and buffer sizes. Additionally, and unlike existing approaches, our framework supports a wide range of trade-offs between multiple design goals such as power, performance, cost and area, during the synthesis process.

(ii) Fast and Accurate Simulation Models: In order to speed up modeling effort and simulation speed during exploration of the increasingly large and complex MPSoC designs, we have developed a novel modeling abstraction called CCATB (Cycle Count Accurate at Transaction Boundaries) that not only allows rapid system prototyping, but also enables fast and accurate simulation for on-chip communication architecture exploration. The CCATB kernel effectively aggregates delays during simulation, reducing event scheduling overhead and speeding up execution. CCATB models are an
order of magnitude faster than existing simulation models, and also require much less time to capture the hardware and software components in an MPSoC design.

(iii) Fast and Accurate Power Estimation Models: To estimate the power dissipation of on-chip communication architectures early in the design flow, we have developed a methodology to generate power models for the various components (logic and wires) of a communication architecture fabric. The power models are close in accuracy to detailed gate-level estimates, while enabling power estimation several orders of magnitude faster than traditional approaches. These models ultimately allow designers to accurately estimate the power overhead of on-chip communication architecture configurations much earlier in the design flow, enabling much more informed decisions during synthesis.

(iv) Physically-aware On-Chip Communication Architecture Synthesis: Since most synthesis techniques do not consider physical implementation issues, there is a vast disconnect between the on-chip communication architecture synthesized by these approaches, and the feasibility of its physical implementation. In order to address this challenge, we have developed a framework that introduces physical awareness into the communication architecture synthesis process. The framework makes use of a high level floorplanner to create an early MPSoC layout, generates routing estimates and uses other heuristics to make the synthesis process more physically aware. Unlike existing techniques, our framework enables automated detection and elimination of clock cycle timing violations during communication architecture synthesis. This enables designers to save weeks to months of costly design iterations to resolve the timing violations at the physical level, later in the design flow.
(v) **Co-synthesis of Memory and Communication Architectures:** Traditionally, memory synthesis is performed before the on-chip communication architecture synthesis step. While treating these two steps separately is done mainly due to tractability issues, it can lead to sub-optimal design decisions. We have developed a novel framework that automatically co-synthesizes memory and communication architectures. Compared to the traditional approach, our framework significantly reduces the number of buses and the memory area, while guaranteeing application performance constraints. This results in a reduction of overall power, cost and area for the MPSoC design, compared to the traditional approach of separate synthesis.

In order to validate these contributions and show their practical feasibility, we applied the various methodologies and frameworks that we developed on industrial-strength MPSoC applications. Wherever possible, we compared our approaches with results from existing work to quantify improvements obtained from the techniques developed in this dissertation.

### 1.5 Dissertation Outline

The rest of the thesis is organized as follows:

- In Chapter 2, we present a fast and accurate system-level modeling abstraction that not only reduces time to model the various hardware and software components in an MPSoC design, but also significantly improves simulation speed, to reduce overall communication architecture exploration time.
• In Chapter 3, we present a methodology for generating power models for on-chip communication architecture power estimation. We show how these models enable fast and accurate power estimation for the communication architecture, early in the design flow.

• In Chapter 4, we describe our novel on-chip communication architecture synthesis framework. This automated framework enables comprehensive communication architecture synthesis, generating topology, component mappings and protocol parameter values, while trading off multiple design constraints such as power, performance, cost and area.

• In Chapter 5, we describe an approach for introducing physical awareness into on-chip communication architecture synthesis, to enable early detection and elimination of timing violations during the synthesis phase.

• In Chapter 6, we present a novel approach for memory-communication architecture co-synthesis that simultaneously synthesizes the memory and communication architectures to reduce overall system cost, area and power dissipation.

• In Chapter 7, we conclude with a summary of our contributions, and directions for future research.
Chapter 2  
CCATB: Fast and Accurate Simulation Models for On-Chip Communication Architecture Exploration

2.1 Introduction

With the increasing SoC design complexity and advances in technology scaling, more and more components (such as CPUs, memories, peripherals, DSPs, etc.) are now being integrated on a single chip, to share the processing load. These components frequently exchange data with each other, which has led to a corresponding increase in overall on-chip communication. Inter-component communication is often in the critical path of a SoC design and is a very common source of performance bottlenecks [8] [9]. It therefore becomes imperative for system designers to focus on exploring the on-chip communication space quickly, reliably and early in the design flow to make the right choices and eliminate performance bottlenecks under time-to-market pressures.

On-chip communication involves numerous sources of delay such as signal propagation along the wires, synchronization (e.g., handshaking) overhead, transfer modes (e.g., pipeline access, burst transfer etc.), arbitration mechanisms for congestion management, cross-bridge transfers and data packing/unpacking at the interfaces. There are two main ways of estimating these delays and estimating performance of on-chip communication architectures: (i) static estimation, and (ii) dynamic (simulation-based) estimation. Static estimation techniques allow for fast performance estimation, but are unable to account for non-deterministic traffic generation by the components on buses,
and consequently cannot predict dynamic component (e.g., memory access) delays as well as dynamic delays arising due to arbitration and traffic congestion, cache misses, burst interruptions, interface buffer overflows and the effect of advanced bus features such as SPLIT transactions [10] and out-of-order (OO) transaction completion [11]. As a result, these static approaches have limited applicability. A more accurate approach to performance estimation requires creating a model of the application that can be simulated. This allows a more accurate estimation of the data traffic behavior on the buses and the corresponding delays can be more reliably assessed. However, selecting the appropriate modeling abstraction that can provide an acceptable balance of simulation speed, accuracy and modeling overhead is a very challenging problem. The modeling abstraction introduced in this chapter provides system designers with a fast and accurate simulation abstraction that has low modeling overhead, for fast and reliable exploration on-chip communication architecture exploration, early in the design flow.

2.1.1 Chapter Overview

In this chapter, we introduce a novel modeling abstraction level called CCATB (Cycle Count Accurate at Transaction Boundaries) for on-chip communication space exploration. CCATB does not maintain accuracy at every cycle boundary, but instead raises the modeling abstraction and maintains cycle count accuracy at transaction boundaries (i.e., the number of bus cycles that elapse at the end of a read/write transaction is the same when compared to the number of cycles elapsed in a detailed cycle accurate model). Our modeling abstraction maintains overall cycle count accuracy needed to gather statistics for accurate communication space exploration, while
optimizing the models for faster simulation. CCATB essentially trades off intra-transaction visibility to gain speedup in simulation and modeling. It allows faster system prototyping and more importantly better simulation performance, while maintaining cycle count accuracy, compared to existing modeling abstractions used for on-chip communication space exploration.

In Section 2.2, we first introduce the communication architecture terminology and standards that we will be using to explain the CCATB abstraction in this chapter. In Section 2.3 we present an overview of the CCATB modeling abstraction for exploring on-chip communication architectures. In Section 2.4, we present an implementation of the CCATB simulation model and illustrate the sources of speedup obtained by our abstraction. In Section 2.5 we describe two exploration case studies where we use CCATB models to explore the communication design space of MPSoC subsystems from the broadband communication and multimedia domains. In Section 2.6, we present experimental studies to compares modeling effort and simulation speeds for the CCATB modeling abstraction with existing modeling abstractions used by designers today.

2.1.2 Related Work

Traditionally, designers had been exploring the communication architecture space at the register transfer level (RTL). With rising design complexity over the last few years, RTL simulation speed has been found to be not only too slow to allow adequate coverage of the large communication design space, but making small changes in the design ends up requiring considerable re-engineering effort due to the highly detailed and complex
nature of RTL models. To overcome the limitations of RTL simulation, system designers have raised the abstraction level of exploration models.

Figure 2.1 Traditional Modeling Abstractions for Communication Space Exploration

Figure 2.1 shows the modeling abstraction levels for communication space exploration that have been proposed and used over the last decade. These abstractions are usually captured with high level languages such as C/C++ [12], and give an early estimate of the system characteristics before committing to RTL development. In Cycle Accurate (CA) models [13] [14], system components and the communication architecture are captured at
a cycle and signal accurate level. While these models are extremely accurate, they are too
time-consuming to model and only provide a moderate speedup over RTL models. Pin-
Accurate Bus Cycle Accurate (PA-BCA) models [15] capture the system at a higher
abstraction level than CA models. Behavior inside components need not be scheduled at
every cycle boundary, which enables rapid system prototyping and considerable
simulation speedup over RTL. The component interface and the bus are still modeled at a
cycle and pin accurate level, which enables accurate communication space exploration.
However, with rising design complexity in modern MPSoC designs, even the simulation
speedup gained with PA-BCA models is not enough.

More recent research approaches have focused on using concepts found in the
Transaction Level Modeling (TLM) to speed up simulation. Transaction Level Models
[16], [17], [18] are bit-accurate models of a system with specifics of the bus protocol
replaced by a generic bus (or channel), and where communication takes place when
components call read() and write() methods provided by the channel interface. Since
detailed timing and pin-accuracy are omitted, these models are fast to simulate and are
useful for early functional validation of the system. Our work in [17] describes how TLM
can be used for early system prototyping and embedded software development. Paulin et
al. [19] define a system level exploration platform for network processors that need to
handle high speed packet processing. The SOCP channel described in their approach is
based on OCP [20] semantics and is essentially a simple TLM channel with a few added
details such as support for split transactions [10]. Nicolescu et al. [21] propose a
component based bottom-up system design methodology where components modeled at
different abstractions are connected together with a generic channel like the one used in
TLM, after encapsulating them with suitable wrappers. Commercial tools such as the Incisive Verification Platform [22], ConvergenSC System Designer [23] and Synopsys System Studio [24] have also started adding support for system modeling at the higher TLM abstraction, in addition to lower level RTL modeling. Finally, the OCP-IP specification [20] describes different layers of abstraction, to capture a communication system at varying levels of detail. Layer-0 is an extremely detailed pin, bit and cycle accurate model; Layer-1 is a transaction-based bus cycle accurate (T-BCA) abstraction; Layer-2 is an approximately timed transaction level model (T-TLM) which uses approximate timing and does not capture bus protocol details; Layer 3 is an untimed transaction-level model (UT-TLM).

Recently, research efforts [25] [26] [27] [28] have focused on adapting TLM concepts to speed up architecture exploration. Zhu et al. [25] use function calls instead of slower signal semantics to describe models of AMBA 2.0 and CoreConnect bus architectures at a high abstraction level. However, the resulting models are not detailed enough for accurate communication exploration. Caldari et al. [26] similarly attempt to model AMBA 2.0 using function calls for reads/writes on the bus, but also model certain bus signals and make extensive use of SystemC clocked threads which can slow down simulation. Ogawa et al. [27] also model data transfers in AMBA 2.0 using read/write transactions but use low level handshaking semantics in the models that need not be explicitly modeled to preserve cycle accuracy. Recently, ARM released the AHB Cycle-Level Interface Specification [28] which provides the definition and compliance requirements for modeling AHB at a cycle-accurate level in SystemC. Function calls are used to replace all bus signals at the interface between components and the bus. Although
using function calls speeds up simulation, there is a lot of opportunity for improvement by reducing the number of calls while maintaining cycle accuracy, as we show later in this chapter.

2.2 On-Chip Communication Architectures: Background

In this section, we first introduce the terminology used in conjunction with on-chip communication architectures. Next we briefly discuss standards used in on-chip communication architectures, and present an overview of the commonly used ARM AMBA communication architecture standard.

2.2.1 Terminology

We begin by reviewing the basic terminology that is used to describe bus-based communication architectures, and for systems which use buses for communication. Figure 2.2 shows a simple System-on-Chip (SoC) design in which several (computational) components are interconnected using a bus-based communication architecture. Components which initiate and control read and write data transfers are referred to as Masters. The Processor and DSP components in Figure 2.2 are an example of master components which read/write data from/to other components in the system. Every master component is connected to the bus using a set of signals which are collectively referred to as a master port. The components that simply respond to data transfer requests from masters (and cannot initiate transfers themselves) are referred to as slaves, and have corresponding slave ports. The three memory blocks in Figure 2.2 are
examples of slaves, which can handle requests for data read and write from other components (e.g., Processor, DSP), but cannot initiate such transfers themselves. The component ports are actually part of its interface with the bus. An interface can be simple, consisting merely of the set of connecting wires to the bus. Or it could be more complex, consisting of buffers, frequency converters etc. in order to improve performance.

![Diagram of a system with a bus-based communication architecture](image)

**Figure 2.2 Example of a System with a Bus-based Communication Architecture**

Some components can have both master and slave ports, which means that they can act as both masters and slaves. These components are master/slave hybrid components. For instance, the DMA (Direct Memory Access) component in Figure 2.2 has a slave port which allows the Processor to write into (and read from) the DMA configuration register.
file, in order to initialize and configure it. Once configured, the DMA component uses its master port to initiate and control data transfers between memory blocks (which would otherwise have been managed by the Processor; as a result the Processor is freed up to perform other activity which typically improves system performance). Similarly, the Memory Controller component has a slave port which is used by the DSP component to initialize and configure its functionality. Once configured, the Memory Controller can initiate and control data transfers with external memory components connected to it, using its master port.

In addition to the wires, a bus-based communication architecture also consists of logic components such as decoders, arbiters and bridges. A decoder is a logic component that decodes the destination address of a data transfer initiated by a master, and selects the appropriate slave to receive the data. It can either be a separate logic component, or integrated into a component interface. An arbiter is a logic component that determines which master to grant access to the bus, if multiple masters request access to the bus simultaneously. Typically, some form of a priority scheme is used, to ensure that critical data transfers in the system are not delayed. Finally, a bridge is a logic component that is used to connect two buses. It can vary in its complexity, depending on whether it interconnects two buses with the same or different protocols, with the same or different clock frequencies etc. A bridge connects to a bus using a master or a slave port, just like any other component. The type of port used to connect to a bus depends on the direction of data transfers passing through it. For instance, in the example shown in Figure 2.2, the DMA and Processor components on Bus 1 initiate and control data transfers to Bus 2 by sending data to the slave port of the bridge on Bus 1, which transfers it to its master port.
on Bus 2 and sends the data to its destination. Since the DSP and Memory Controller do not initiate and control data transfers to components on Bus 1, a single bridge is sufficient as shown in Figure 2.2. However, if these components needed to transfer data to Bus 1, another bridge with a slave port on Bus 2 and a master port on Bus 1 would be required.

2.2.2 On-Chip Communication Architecture Standards

SoC designs typically have several different types of components such as processors, memories, custom hardware, peripherals and external interface IP (Intellectual Property) blocks that need to communicate with each other. In SoC design houses, some of these components might be designed from scratch, while others are usually reused from previous designs or procured from external IP vendors. Each of these components has an interface to the outside world consisting of a set of pins that are responsible for sending/receiving addresses, data and control information to/from other components. The choice of pins at the interface is governed by the particular bus protocol that the component connects to. In order to seamlessly integrate all these components into a SoC design, it is necessary to have some kind of a standard interface definition for the components. Without a standard interface definition, the component interfaces will not be compatible with the bus architecture implementation, and consequently not function correctly. In such a scenario, the components will require the design of logic wrappers at their interfaces to correctly interface with the bus architecture being used, which can be a very time consuming activity that can take from several weeks to months (for the design and its verification).
Over the past decade and a half, several bus-based on-chip communication architecture standards have been proposed to speed up SoC integration and promote IP reuse over several designs. These standards include AMBA 2.0 [10] and 3.0 [11], IBM CoreConnect [29], STMicroelectronics STBus [30], Sonics SMART Interconnect [31], Opencores WISHBONE [32], and Altera Avalon [33]. A communication architecture standard defines a specific data transfer protocol, which in turn decides the number and functionality of the pins at the interface of the components. Usually, bus-based communication architecture standards define the interface between components and the bus architecture, as well as the bus architecture that implements the data transfer protocol. One of the most commonly used on-chip communication architecture standards is the AMBA architecture standard. Since we use the AMBA architecture to describe and demonstrate the features and exploration capabilities of CCATB in subsequent sections, we give a brief overview of the standard here.

The AMBA 2.0 bus architecture consists of the AHB (Advanced High-performance Bus), APB (Advanced Peripheral Bus) and ASB (Advanced System Bus) buses. The AHB bus is used for high bandwidth and low latency communication, primarily between CPU cores, high performance peripherals, DMA controllers, on-chip memories and interfaces such as bridges to the slower APB bus. The APB is used to connect slower peripherals such as timers, UARTs etc. and uses a bridge to interface with the AHB. It is a simple bus that does not support the advanced features of the AHB bus. The ASB bus is an earlier version of the high-performance bus that has been superseded by AHB in current designs. More recently, ARM released the AMBA 3.0 [11] standard with the next generation of high-performance bus protocol called the Advanced eXtensible Interface
(AXI). We give a brief overview of the main features of the high performance bus protocols in AMBA.

**AMBA 2.0 AHB**

The Advanced High-Performance Bus (AHB) is a high-speed, high-bandwidth bus that supports multiple masters. AHB supports a multi-layer bus architecture to optimize system bandwidth and improve performance. It supports pipelined operations for high speed memory and peripheral access without wasting precious bus cycles. Burst transfers allow optimal usage of memory interfaces by giving advance information of the nature of the transfers. AHB also allows split transactions which maximize the use of the system bus bandwidth by enabling high latency slaves to release the system bus during the dead time while the slave is completing its transaction. In addition, wide bus configurations from 32 up to 1024 bits wide are supported.

**AMBA 3.0 AXI**

The Advanced eXtensible Interface (AXI) has all the advanced features of the AHB bus such as pipelined and burst transfers, multi-master configuration and a wide data bus. In addition, it has support for separate read and write channels, unaligned data transfer using byte strobes and improved burst mode operation (only the start address of the burst is broadcast on the address bus). An important feature of AXI is support for multiple outstanding transactions and out-of-order (OO) transaction completion. OO transaction completion allows slaves to relinquish control of the bus, complete received transactions in any order and then request for re-arbitration so a response can be sent back to the
master for the completed transaction. This can dramatically boost performance in high performance systems by allowing better utilization of shared buses. AXI also provides enhanced protection support (secure/non-secure transactions), enhanced system cache/buffer support (pins for specifying write-back/write through attributes and allocation strategies), a FIXED burst mode (for repeated access to the same location) and exclusive access support for semaphore type operations.

2.3 CCATB Overview

As the previous section indicated, bus architectures such as AMBA have several parameters that can be configured to improve performance. Our goal is to improve simulation performance for reliable exploration of on-chip communication architectures as early as possible in the design flow.

2.3.1 Modeling Abstraction

To enable fast exploration of the communication design space, we introduce a novel modeling abstraction level that is ‘cycle accurate’ when viewed at ‘transaction boundaries’. For this reason we call our model *Cycle Count Accurate at Transaction Boundaries (CCATB)* [110] [111] [112] [113]. A transaction in this context refers to a *read* (or *write*) operation issued by a master to a slave, that can either be a single data word or a multiple data burst transfer. Transactions at the CCATB level are similar to transactions at the TLM level [17] except that we additionally pass bus protocol specific control and timing information. Unlike PA-BCA and T-BCA models, we do not maintain
accuracy at every cycle boundary. Instead, we raise the modeling abstraction and maintain cycle count accuracy at transaction boundaries, i.e., the number of bus cycles that elapse at the end of a transaction is the same when compared to cycles elapsed in a detailed cycle/pin accurate system model. A similar concept was proposed by Bergamaschi and Raje [34] where Observable Time Windows were defined and used for verifying results of high level synthesis. We maintain overall cycle count accuracy needed to gather statistics for accurate communication space exploration, while optimizing the models for faster simulation. Our approach essentially trades off intra-transaction visibility to gain simulation speedup.

2.3.2 Modeling Language

We chose SystemC [17] [18] to capture designs at the CCATB abstraction level, as it provides a rich set of primitives for modeling communication and synchronization - channels, ports, interfaces, events, signals and wait-state insertion. Concurrent execution is performed by multiple threads and processes (lightweight threads) and execution schedule is governed by the scheduler. SystemC also supports capture of a wide range of modeling abstractions from high level specifications to pin and timing accurate system models. Since it is a library based on C++, it is object oriented, modular and allows data encapsulation – all of which are essential for easing IP distribution, reuse and adaptability across different modeling abstraction levels. It should be noted however, that CCATB is ‘language agnostic’ and can be modeled with any high level modeling language.
2.3.3 Integrating CCATB into an MPSoC Design Flow

We define a modeling methodology which integrates our CCATB model in a high level system design flow. Figure 2.3 depicts our proposed flow which has five system models at different abstraction levels.

![Figure 2.3 System Design Flow with CCATB](image)

At the topmost level is a *specification* model which is a high level algorithmic implementation of the functionality of the system. This model is generally captured in C or C++ and is independent of the hardware architecture that would eventually be used to
implement the algorithm. After selecting available hardware components and partitioning functionality between hardware and software, we arrive at the TLM model ported to SystemC. At this level, high level functional blocks representing hardware components such as CPUs, memories and peripherals are connected together using a bus architecture-independent generic channel. This system model is used for early embedded software development and high-level platform validation. It is generally untimed, but the model can be annotated with some timing information if a high level estimate of system performance is required. Once the bus architecture is decided, the channels are merged onto a bus topology, each bus is annotated with timing and protocol details, and the interface is refined to obtain the CCATB model. This model is used for fast communication space and system performance exploration. The read(), write() channel interface from the TLM level remains the same as explained earlier – except that now bus-architecture specific control information also needs to be passed. Components from the TLM level can be easily and quickly refined to add this detail. If observable cycle accuracy for early system debugging and validation is required, the read() and write() interface calls can be decomposed into function calls which consist of bus pins in the PA-BCA model. This is a bus cycle accurate model, where the state of the system on the bus is accurately observable at every cycle boundary, instead of only at transaction boundaries like in the CCATB model. Finally, the components are refined further to obtain pin/cycle-accurate models which can be manually or automatically mapped to RTL, or simply be used to co-simulate with existing RTL components for better simulation performance while validating system design at a low level.
2.3.4 Component Model Characteristics

Bus architectures in CCATB are modeled by extending the generic TLM channel [17] to include bus architecture specific timing and protocol details. Arbiter and decoder modules are integrated with this channel model. Computation blocks (masters and slaves) are modeled at the behavioral abstraction level.

Figure 2.4 CCATB Transaction Example

Masters are active blocks with possibly several computation threads and ports to interface with buses. One of our goals was to keep a consistent interface when refining models from the TLM level down to our CCATB level (Figure 2.3). Figure 2.4 shows the interface used by the master to communicate with a slave. In the figure, *port* specifies the port to send the read/write request on (since a master may be connected to multiple buses). *addr* is the address of the slave to send the transaction to. *token* is a structure that contains pointers to data and control information. Table 2.1 shows the main fields in this
token data structure passed by the master and received by the arbiter. The *status* field in the token structure contains the status of the transaction, as returned by the slave. At the TLM level, since the bus is modeled as an abstract channel without including any specific details of the bus protocol, the *data_cntrl* structure contains just the *m_data*, *m_burst_length* and *m_byte_enable* fields. The other fields are specific to bus protocols and are thus omitted since we are only concerned with transferring data packets from the source to its destination at this level. Thus, when we refine a master IP from the TLM level to the CCATB level, the only change is to set protocol specific parameters before calling the interface functions.

### Table 2.1 Fields in token structure

<table>
<thead>
<tr>
<th>Request field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_data</td>
<td>pointer to an array of data</td>
</tr>
<tr>
<td>m_burst_length</td>
<td>length of transaction burst</td>
</tr>
<tr>
<td>m_burst_type</td>
<td>type of burst (incr, fixed, wrapping etc.)</td>
</tr>
<tr>
<td>m_byte_enable</td>
<td>byte enable strobe for unaligned transfers</td>
</tr>
<tr>
<td>m_read</td>
<td>indicates whether transaction is read/write</td>
</tr>
<tr>
<td>m_lock</td>
<td>lock bus during transaction</td>
</tr>
<tr>
<td>m_cache</td>
<td>cache/buffer hints</td>
</tr>
<tr>
<td>m_prot</td>
<td>protection modes</td>
</tr>
<tr>
<td>m_transID</td>
<td>transaction ID (needed for OO access)</td>
</tr>
<tr>
<td>m_busy_idle</td>
<td>schedule of busy/idle cycles from master</td>
</tr>
<tr>
<td>m_ID</td>
<td>ID for identifying the master</td>
</tr>
<tr>
<td>status</td>
<td>status of transaction (returned by slave)</td>
</tr>
</tbody>
</table>

Slaves are passive entities, activated only when triggered by the arbiter on a request from the master, and have a register/memory map to handle read/write requests. The arbiter calls *read()* and *write()* functions implemented in the slave, as shown for the SDRAM controller in the figure. An excerpt of the read function from a memory controller is shown in Figure 2.4. Slaves can also have optional (lightweight) processes.
triggered by SystemC *events*, to perform computation if needed. The functionality of the slave IP remains unchanged when refining the model from the TLM level to the CCATB level, unless the slave IP supports special bus protocol specific features such as having an outstanding instruction queue for out-of-order transaction completion in the AXI protocol, in which case these details need to be added.

In accordance with the principle of Interface Based Design [35], preexisting master and slave IP modules with different interfaces can be incorporated in the model using an adapter written in SystemC. For instance, we used adapter code written in SystemC in our exploration environment to interface ARM processor ISS models (which are not written in SystemC) with the TLM/CCATB SystemC interface.

### 2.3.5 Maintaining Cycle Count Accuracy with CCATB

We will now illustrate how the CCATB model maintains cycle count accuracy at transaction boundaries for different call sequences of the AMBA 2.0 protocol. First, we will describe the AMBA 2.0 signals used in these examples, and then go into the details of the examples shown in Figures 2.5 (a), (b) and (c).

In AMBA 2.0, when a master needs to send or receive data, it requests the arbiter for access to the bus by raising the *HBUSREQx* signal. The arbiter, in turn, responds to the master via the *HGRANTx* signal. Depending on which master gains access to the bus, the arbiter drives the *HMASTERx* signals to indicate which master has access to the bus (this information is used by certain slaves). When a slave is ready to be accessed by a master, it drives the *HREADYx* signal high. Only when a master has received a bus grant from the arbiter via *HGRANTx* and detects a high *HREADY* signal from the destination slave, will
it initiate the transaction. The transaction consists of the master driving the $HTRANS_x$ signal, which describes the type of transaction (sequential or non-sequential), the $HADDR_x$ signals which are used to specify the slave addresses, and $HWDATA_x$ if there is write data to be sent to the slave. Any data to be read from the slave appears on the $HRDATA_x$ signal lines. The master also drives control information about the data transaction on other signal lines – $HSIZE_x$ (size of the data item being sent), $HBURST_x$ (number of data items being transferred in a burst transaction), $HWRITE$ (whether the transfer is a read or a write) and $HPROT_x$ (contains protection information for slaves which might require it).
Figure 2.5 Reference AMBA 2.0 call sequences
We now describe the examples shown in Figure 2.5. In the first example in Figure 2.5 (a), a master requests an incremental write burst of length 4 data packets and the arbiter immediately grants it access to the bus. The transaction is initiated and data sent to the slave, but before it can process the final data packet in the sequence, the slave needs to perform involved computation with the previously written data that takes up 2 cycles. For this duration, the slave drives the HREADY signal low to indicate to the master that it is not ready yet to receive the final data packet in the burst. The burst transaction resumes...
once the slave drives $HREADY$ high. The sequence of actions in the CCATB model is shown in Figure 2.6. The arbiter accounts for the request ($REQ$) and arbitration ($ARB$) delays for the write request before invoking the slave to complete the transaction. The slave performs the write and returns a token structure which contains the status of the write and an indication to the arbiter that 2 wait states need to be inserted. The arbiter then increments simulation time with the slave delay ($SLV$), burst length ($BURST_LEN$) and pipeline startup ($PPL$) delays. The arbiter then returns the status of the writes at the end of the transaction to the master.

Figure 2.5 (b) illustrates a similar scenario, but in this case there is a delay in generating the data at the master end instead of a processing delay at the slave end. After the write burst initiates, the master indicates that it requires extra cycles to generate write data for the slave by sending a $BUSY$ status on the $HTRANS[1:0]$ lines. In the CCATB model, the arbiter gets a schedule of busy cycles from the master when it receives the transaction request, and thus it accounts for the $BUSY$ cycle delay in the transaction, along with the other delays discussed above. There is no delay at the slave and consequently no increment in simulation time due to slave delay in this case.

In Figure 2.5 (c), after a master requests access to the bus for a write burst, another master requests the bus for a write burst. While there is no delay at the master or the slave end for the first write burst, there is delay in generating the data at the master end for master M2, which is indicated by the $BUSY$ status on the $HTRANS[1:0]$ lines. In the CCATB model, the arbiter accounts for the $REQ$, $ARB$, $BURST_LEN$ and $PPL$ delays and increments simulation time. For the subsequent transaction by master M2, the request has already been registered at the arbiter and no arbitration is required, so there is no $REQ$ or
Since transfers are pipelined, there is also no pipeline startup delay like in the case of master M1. Thus there is no PPL delay. There is however delay which is dependent on the burst length (BURST_LEN) and the busy cycles (BUSY) which is accounted for by the arbiter. Like in the previous scenario, the slave does not delay either of the burst transactions, so there is no simulation time increment due to slave delay.

### 2.4 Simulation Speedup using CCATB

We now describe an implementation of the CCATB simulation model to explain how we obtain simulation speedup. We consider a design with several bus subsystems each with its own separate arbiter and decoder, and connected to the other subsystems via bridges. The bus subsystem supports pipelining, burst mode transfers and out-of-order (OO) transaction completion which are all features found in high performance bus architectures such as AMBA 3.0 AXI.

We begin with a few definitions. Each bus subsystem is characterized by a tuple set $X$, where $X = \{R_{\text{pend}}, R_{\text{act}}, R_{\text{oo}}\}$. $R_{\text{pend}}$ is a set of read/write requests pending in a bus subsystem, waiting for selection by the arbiter. $R_{\text{act}}$ is a set of read/write requests actively executing in a subsystem. $R_{\text{oo}}$ is a set of out-of-order read/write requests in a subsystem that are waiting to enter into the pending request set ($R_{\text{pend}}$) after the expiration of their OO latency period (number of cycles that elapse after the slave releases control of the bus and before it requests for re-arbitration). As mentioned previously in Section 2.2, OO transaction completion allows slaves to relinquish control of the bus, complete received transactions in any order and then request for re-arbitration so a response can be sent
back to the master for the completed transaction. We define $A$ to be a superset of the sets $X$ for all $p$ bus subsystems in the entire system,

$$A = \bigcup_{i=1}^{p} X_i$$

Next we define $\tau$ to be a transaction request structure, issued by a master to a slave. In addition to the subfields in Table 2.1, it also includes the following subfields:

- $\text{wait\_cyc}$ – specifies the number of wait cycles before the bus can signal transaction completion to the master
- $\text{oo\_cyc}$ – specifies the number of wait cycles before the request can apply for re-arbitration at the bus arbiter
- $\text{ooflag}$ – indicates if the request is an out-of-order transaction

Let $\text{status}$ be defined as a transaction response structure returned by the slave. It contains a field ($\text{stat}$) that indicates the status of the transaction (OK, ERROR etc.) as well as fields for the various delays encountered such as those for the slave interface ($\text{slave\_int\_delay}$), slave computation ($\text{slave\_comp\_delay}$) and bridges ($\text{bridge\_delay}$).

Finally, let $M = \{m_1, m_2, ..., m_n\}$ be a set of all masters in the system. Each master is represented by a value in this set which corresponds to the sum of (i) the number of cycles before the next read/write request is issued by the master and (ii) the master interface delay cycles. These values are maintained in a global table with an entry for each master and do not need to be specified manually by a designer – a preprocessing...
stage can automatically insert directives in the code to update the table at the point when a master issues a request to a bus.

Figure 2.7 CCATB simulation flow

Our approach speeds up simulation by preventing unnecessary invocation of simulation components and efficiently handling idle time during simulation. We now describe the implementation for our simulation model to show how this is accomplished. Figure 2.7 gives a high level overview of the flow for the CCATB simulation model. There are two main phases in the model – the first is triggered on a positive edge of the system clock while the second is triggered on the negative edge of the system clock. In the first phase, on the positive edge of the clock, we gather all the read and write requests in the system (GatherRequests). In the second phase, on the negative edge of the clock,
we process these requests by calling the \textit{HandleBusRequests} procedure, which in turn calls various functions and sub-procedures to perform different tasks. In the first step inside the \textit{HandleBusRequests} procedure, we handle completed requests and notify masters about completed transactions (\textit{HandleCompletedRequests}). In the second step, we arbitrate on the buses in the system to select the requests which have the highest priority and grant them access to their respective buses (\textit{ArbitrateRequest}). In the third step, we issue the selected \textit{read} or \textit{write} requests to the slaves, receive a response from the slaves (\textit{issue}) and update the request tokens with appropriate delay cycles for which we must wait before notifying the master (\textit{UpdateDelaysAndSets}). In the fourth step we determine the number of cycles to increment the simulation time (\textit{DetermineIncrementPeriod}) before finally incrementing the simulation time (\textit{IncrementSimulationTime}) in the fifth step. This completes one iteration cycle of the simulation (which could represent one or more \textit{simulation} cycles based on the amount by which simulation time is incremented) which is repeated till the simulation ends.

\begin{verbatim}
procedure GatherRequests()
begin
  if request then
    \(\tau \leftarrow \text{request}\)
    \(\tau.\text{wait}\_\text{cyc} \leftarrow 0\)
    \(\tau.\text{oo}\_\text{cyc} \leftarrow 0\)
    \(\tau.\text{ooFlag} \leftarrow \text{FALSE}\)
    \(R_{\text{pend}} \leftarrow R_{\text{pend}} \cup \tau\)
  end
\end{verbatim}

\textbf{Figure 2.8 GatherRequests procedure}

We now describe the implementation of our simulation model in detail. Figure 2.8 shows the \textit{GatherRequests} procedure from a bus module, which is triggered on a positive
clock edge, for every read or write transaction request issued by a master connected to that bus. GatherRequests simply adds the transaction request to the set of pending requests $R_{\text{pend}}$ for the bus subsystem.

```
procedure HandleBusRequests()
begin
  for each set $X \in \text{A do}$
    HandleCompletedRequests($R_{\text{pend}}, R_{\text{act}}, R_{\text{oo}}$)
    $T \leftarrow \text{ArbitrateRequest}(X, R_{\text{pend}})$
    for each request $\tau \in T$ do
      if ($\tau.\text{ooflag} \equiv \text{TRUE}$) then
        $R_{\text{act}} \leftarrow R_{\text{act}} \cup \tau$
      else
        status $\leftarrow \text{issue}(\tau.\text{port}, \tau.\text{addr}, \tau)$
        UpdateDelaysAndSets(status, $\tau, R_{\text{act}}, R_{\text{oo}}$)
    $\psi \leftarrow \text{DetermineIncrementPeriod}(A)$
    IncrementSimulationTime($\psi, A, M$)
end
```

**Figure 2.9 HandleBusRequests procedure**

```
procedure HandleCompletedRequests($R_{\text{pend}}, R_{\text{act}}, R_{\text{oo}}$)
begin
  $S_{\text{pend}} \leftarrow R_{\text{pend}}$ ; $S_{\text{act}} \leftarrow \text{null}$ ; $S_{\text{oo}} \leftarrow \text{null}$ ;
  for each request $\tau \in R_{\text{act}}$ do
    if ($\tau.\text{wait}_\text{cyc} \equiv 0$) then
      notify($\tau.\text{port}, \tau.\text{addr}, \tau$)
    else
      $S_{\text{act}} \leftarrow S_{\text{act}} \cup \tau$
  for each request $\tau \in R_{\text{oo}}$ do
    if ($\tau.\text{ooc}_\text{cyc} \equiv 0$) then
      $S_{\text{pend}} \leftarrow S_{\text{pend}} \cup \tau$
    else
      $S_{\text{oo}} \leftarrow S_{\text{oo}} \cup \tau$
  $R_{\text{pend}} \leftarrow S_{\text{pend}}$ ; $R_{\text{act}} \leftarrow S_{\text{act}}$ ; $R_{\text{oo}} \leftarrow S_{\text{oo}}$
end
```

**Figure 2.10 HandleCompletedRequests procedure**

On the negative clock edge, the HandleBusRequests procedure (Figure 2.9) in the bus module is triggered, and calls several sub-procedures and functions to handle the
communication requests in the system. *HandleBusRequests* first calls the *HandleCompletedRequests* sub-procedure (Figure 2.10) for a bus subsystem $X$, to check if any executing requests in $R_{act}$ have completed, in which case the appropriate master is notified and the transaction completed. *HandleCompletedRequests* also removes an out-of-order request from the set of out of order requests $R_{oo}$ and adds it to the pending request set $R_{pend}$ if it has completed waiting for its specified *OO latency period*.

```
function ArbitrateRequest(\(X, R_{pend}\))
begin
    \(T \leftarrow null\)
    for each independent channel \(c \in X\) do
        \(T \leftarrow T \cup \text{ArbitrateOnPolicy}(c, R_{pend})\)
    \(R_{pend} \leftarrow R_{pend} \setminus T\)
    return \(T\)
end
```

**Figure 2.11 ArbitrateRequest procedure**

Next, we arbitrate to select requests from the pending request set $R_{pend}$ which will be granted access to a bus in a bus subsystem $X$. The function *ArbitrateRequest* (Figure 2.11) performs the selection based on the arbitration policy selected for every bus. We assume that a call to the *ArbitrateOnPolicy* function applies the appropriate arbitration policy and returns the selected requests for the bus. After the selection we update the set of pending requests $R_{pend}$ by removing the requests selected for execution (and hence not ‘pending’ anymore). Since a bus subsystem can have independent read and write channels, there can be more than one active request executing in the subsystem, which is why *ArbitrateRequest* returns a set of requests and not just a single request for every subsystem.
After the call to _ArbitrateRequest_ (Figure 2.11) from _HandleBusRequests_ (Figure 2.9), if the _ooflag_ field of the selected request is TRUE, it implies that this request has already been issued to the slave and now needs to wait for \( \tau.wait\_cyc \) cycles before returning a response to the master. Therefore we simply add it to the executing requests set \( R_{act} \). Otherwise we _issue_ the request to the slave which completes the transaction in zero-time and returns a status to the bus module. We use the returned _status_ structure to update the transaction status by calling the _UpdateDelaysAndSets_ procedure.

```
procedure UpdateDelaysAndSets(status, \( \tau \), \( R_{act} \), \( R_{oo} \))
begin
  if (status.stat == OK) then
    \( \tau.status = OK \)
  if (status.oo == TRUE) then
    \( \tau.ooflag \equiv TRUE \)
    \( \tau.oo\_cyc \equiv status.(oo\_delay + slave\_int\_delay\)+
      + slave\_comp\_delay + bridge\_delay)\)+
      + \( \tau.arb\_delay \)
    \( \tau.wait\_cyc \equiv \tau.(busy\_delay + burst\_length\_delay\)+
      + ppl\_delay + bridge\_delay + arb\_delay)\)
    \( R_{oo} \equiv R_{oo} \cup \tau \)
  else
    \( \tau.wait\_cyc \equiv status.(slave\_int\_delay\)+
      + slave\_comp\_delay + bridge\_delay)\)+
      + \( \tau.busy\_delay + burst\_length\_delay\)+
      + ppl\_delay + arb\_delay)\)
    \( R_{act} \equiv R_{act} \cup \tau \)
  end

Figure 2.12 UpdateDelaysAndSets procedure
```

Figure 2.12 shows the _UpdateDelaysAndSets_ procedure. In this procedure we first check for the returned error status. If there is no error, then depending on whether the request is an out-of-order type or not, we update \( \tau.oo\_cyc \) with the number of cycles to
wait before applying for re-arbitration, and $\tau.wait_{cyc}$ with the number of cycles before returning a response to the master. We also update the set $R_{act}$ with the actively executing requests and $R_{oo}$ with the OO requests. If an error occurs, then the actual slave computation delay can differ and is given by the field $error_{delay}$. The values for other delays such as burst length and busy cycle delays are also adjusted to reflect the truncation of the request due to the error.

```plaintext
function DetermineIncrementPeriod(A) begin
    $\psi \leftarrow \infty$
    for each set $X \in A$ do
        for each set $R_{pend} \in X$ do
            if $R_{pend} \neq \text{NULL}$ then
                $\psi \leftarrow 1$
                return $\psi$
        for each set $R_{act} \in X$ do
            for each request $\tau. \in R_{act}$ do
                $\psi \leftarrow \min \{ \psi, \tau.wait_{cyc} \}$
        for each set $R_{oo} \in X$ do
            for each request $\tau. \in R_{oo}$ do
                $\psi \leftarrow \min \{ \psi, \tau.oo_{cyc} \}$
    for each value $\lambda \in M$ do
        $\psi \leftarrow \min \{ \psi, \lambda \}$
    return $\psi$
end
```

Figure 2.13 DetermineIncrementPeriod procedure

After returning from the $UpdateDelaysAndSets$ procedure in $HandleBusRequests$ (Figure 2.9), we find the minimum number of cycles ($\psi$) before we need to iterate through the simulation flow (Figure 2.7) again (i.e. invoke the $GatherRequests$ and $HandleBusRequests$ procedures again), by calling the $DetermineIncrementPeriod$ function (Figure 2.13). This function returns the minimum value out of the wait cycles for every executing request ($\tau.wait_{cyc}$), out-of-order request cycles for all waiting OO requests ($\tau.oo_{cyc}$) and the next request latency cycles for every master ($\lambda$). If there is a
pending request which needs to be serviced in the next cycle, the function returns 1, which is the worst case return value.

```plaintext
procedure IncrementSimulationTime( ψ, A, M )
begin
  for each set X ∈ A do
    for each request τ ∈ R_oo do
      τ.oo_cyc ← τ.oo_cyc - ψ
    for each request τ ∈ R_oo do
      τ.wait_cyc ← τ.wait_cyc - ψ
    for each value λ ∈ M do
      λ ← λ - ψ
      simulation_time ← simulation_time + ψ
  end
Figure 2.14 IncrementSimulationTime procedure
```

Finally, the increment value ψ returned by DetermineIncrementPeriod in HandleBusRequests (Figure 2.9) is used to update simulation time by calling the IncrementSimulationTime procedure, shown in Figure 2.14. By default, the simulation flow incorporating the GatherRequests and HandleBusRequests procedures (shown in Figure 2.7) is invoked for every simulation cycle, but if we find a value of ψ which is greater than 1, we can safely increment system simulation time by that value, preventing unnecessary iterations of the simulation flow and unnecessary invocations of procedures, thus speeding up simulation.

It should be noted that for some very high performance designs it is possible that there is very little scope for this kind of speedup. Although this might appear to be a limitation, there is still substantial speedup achieved over T-BCA and PA-BCA models because we handle all the delays in a transaction in one place – in the bus module, without repeatedly
invoking other parts of the system on every cycle (master and slave threads and processes) which would otherwise contribute to simulation overhead.

In the next two sections, we will present some experiments with the CCATB modeling abstraction. Section 2.5 presents exploration case studies with a broadband communication and a multimedia SoC design. The experiments on these two case studies show how various aspects of the communication architecture design space affect system performance, and why it is important to consider them during a design space exploration effort. This motivates the need for a modeling abstraction such a CCATB that can (i) quickly capture the communication architecture design, (ii) accurately capture delays and parameters of the communication architecture which impact system performance and (iii) allow fast simulation speeds for comprehensive design space exploration. Subsequently in Section 2.6, we change the focus to a comparison of the CCATB approach with other approaches used for communication architecture exploration, such as PA-BCA and T-BCA. The experiments in this section compare modeling time and simulation speedup, and show how CCATB is a suitable abstraction for fast communication architecture design space exploration.

2.5 Design Space Exploration Case Studies

To demonstrate the effectiveness of exploration with CCATB, we present two case studies where we used CCATB models to explore the communication design space of the system. In the first case study we compare and configure bus architectures for a SoC subsystem used in the broadband communication domain. In the second, we assume that
the choice of bus architecture has already been made and we explore different configurations of the bus architecture for a multimedia SoC subsystem.

![Diagram of Broadband Communication SoC platform](image.png)

**Figure 2.15 Broadband Communication SoC platform**

### 2.5.1 Case Study 1: Broadband Communication SoC

In this case study we modeled an actual industrial strength SoC platform and performed several communication space exploration experiments on it. We present four of these in this section. All of these experiments were reproduced and verified at the more refined PA-BCA level. Figure 2.15 shows this SoC platform which has applications in the broadband communication domain. We execute three proprietary benchmarks (*COMPLY, USBDRV* and *SWITRN*) on the ARM926EJ-S processor instruction-set simulator (ISS), each of which activate different modes of operation for the platform. *COMPLY* configures the USB, switch and DMA modules to drive traffic on the shared bus. *USBDRV* also configures the USB and DMA to drive traffic normally on the bus but
the switch activity is restricted. SWITRN configures the switch to drive traffic on the bus normally but restricts USB and DMA activity.

![Figure 2.16 Bus protocol comparison](image1)

![Figure 2.17 Arbitration strategy comparison](image2)

In our first experiment, we attempted to observe the effect of changing communication protocol on overall system performance. We first simulated the platform with the AMBA 2.0 AHB system bus and then replaced it with the AMBA 3.0 AXI bus protocol, keeping the same driver application in both cases and without changing any bus parameters such as arbitration strategy. Figure 2.16 shows that the AXI protocol improves overall system
throughput compared to AHB. This is because in AMBA 2.0, the address bus is occupied mostly by transmission of addresses of transactions within a burst. In contrast, only the first address of a burst is transmitted in AMBA 3.0 AXI, which coupled with transaction reordering allows improved simultaneous read/write transaction execution and better throughput. Our model allows rapid plug-and-play exploration of different bus architectures, requiring changes in just a few lines of code to declare and instantiate the bus in the top-level file.

Next, we explore the effect of arbitration strategies on system performance. We used the AMBA 2.0 AHB system bus and tested the following arbitration strategies - static priority (SP), random priority (RP), round robin (RR), time division multiple access or TDMA with 2 slots for the USB host and 1 for the rest (TDMA1), TDMA with 2 slots for the switch subsystem and 1 for the rest (TDMA2), TDMA1 with RR (TDMA1/RR) and TDMA2 with RR (TDMA2/RR), where the RR strategy is applied only if the selected master has no transaction to issue. Figure 2.17 shows the bus throughput for the three benchmarks. It can be seen that TDMA1/RR outperforms other schemes for COMPLY, while static priority works best for USBDRV (with the USB host given the maximum priority) and SWITRN (where the switch subsystem is given the maximum priority). We measure overall bus throughput – however if bandwidth constraints for certain masters need to be met and overall throughput is a less important criteria, then other strategies might give better results. Also, more involved strategies such as a dynamic priority scheme can be easily introduced into this framework if traffic based adaptable behavior is preferred.
Figure 2.18 Topology configuration A

Figure 2.19 Topology configuration B
To determine the influence of bus hierarchy on improving system performance by eliminating conflicts on a shared bus, we decomposed the shared bus into two hierarchical buses in our next experiment – in configuration A (Figure 2.18) we kept the ARM CPU and DMA master on one bus and the switch subsystem and USB host master on the other. In configuration B (Figure 2.19) we kept the ARM CPU, DMA and the switch subsystem on one bus while the USB host was given a dedicated bus. We used the TDMA1/RR strategy for conflict resolution. Figure 2.20 shows bus conflicts for these cases. It can be seen that configuration A has the least conflicts for COMPLY and SWITRN. This is because configuration A avoids conflicts between the DMA and the switch module which is the main source of conflict in SWITRN and one of the main ones in COMPLY (along with the USB-switch conflict). Configuration B is the best for USBDRV since conflicts between the USB (which drives the maximum traffic) and the DMA (which also drives a lot of traffic) are reduced when the USB is given a dedicated bus.

Finally, we study the effect of changing outstanding request queue size for the SDRAM IF module which supports out-of-order execution of read/write requests as specified by the AMBA 3.0 AXI protocol. Figure 20 shows the effect of change in performance when the queue size is changed. It can be seen that performance saturates and no more gain can be obtained after the queue size has been increased to 4 for COMPLY, and 6 for SWITRN and USBDRV. This is a limit on the number of simultaneous requests issued at any given time for the SDRAM IF by the masters in the system for these benchmarks. It can be seen that this parameter is highly application
dependent and changes with changing application requirements, demonstrating the need for this type of an exploration environment.

![Conflicts (%)](image)

**Figure 2.20 Topology configuration comparison**

![Transactions (read/write) / sec](image)

**Figure 2.21 Varying SDRAM OO Queue size**

### 2.5.2 Case Study 2: Multimedia SoC Subsystem

For our second case study, we explore a consumer multimedia SoC subsystem which performs audio and video encoding for popular codecs such as MPEG. Figure 2.22 shows this platform, which is built around the AMBA 2.0 communication architecture. The system has an ARM926EJ-S processor with embedded software running on it to supervise flow control and perform encryption, a fast USB interface, on-chip memory
modules, a DMA controller, an SDRAM controller to interface with external memory components and standard peripherals such as a timer, UART, interrupt controller, general purpose I/O and a Compact Flash card interface.

Consider a scenario where the designer wishes to extend the functionality of this encoder system to add support for audio/video decoding and an additional AVLink interface for streaming data. The final architecture must also meet peak bandwidth constraints for the USB component (480Mbps) and the AVLink controller interface (768Mbps). Figure 2.23 (a) shows the system with the additional components added to the AHB bus. To explore the effects of changing communication architecture topology and arbitration protocols on system performance, we modeled the SoC platform at the CCATB level and simulated a test program for several interesting combinations of topology and arbitration strategies. For each configuration, we determined if bandwidth constraints were being met and iteratively modified the architecture till all the constraints were satisfied.
Figure 2.23 SoC Communication Architecture Topologies
Table 2.2 Execution cycle counts (in millions of cycles)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>T-TLM</th>
<th>RR</th>
<th>TDMA 1</th>
<th>TDMA 2</th>
<th>SP1</th>
<th>SP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arch1</td>
<td>6.57</td>
<td>27.24</td>
<td>24.65</td>
<td>25.06</td>
<td>25.72</td>
<td>26.49</td>
</tr>
<tr>
<td>Arch2</td>
<td></td>
<td>24.98</td>
<td>23.86</td>
<td>23.03</td>
<td>23.52</td>
<td>23.44</td>
</tr>
<tr>
<td>Arch3</td>
<td></td>
<td>24.73</td>
<td>23.74</td>
<td>22.96</td>
<td>23.11</td>
<td>23.05</td>
</tr>
<tr>
<td>Arch4</td>
<td></td>
<td>22.02</td>
<td>21.79</td>
<td>21.65</td>
<td>21.18</td>
<td>21.26</td>
</tr>
</tbody>
</table>

Table 2.2 shows the system performance (total cycle count for test program execution) for some of the architectures we considered, shown in Figure 2.23 (a), (b), (c) and (d). Column 2 in Table 2.2 shows the performance estimate for a high level approximate-timed transaction-level model (T-TLM) of the system for comparison purposes, which uses behavioral components annotated with delays just like the CCATB model, but uses bus protocol independent *channels* with no contention for communication. This T-TLM model incidentally corresponds to the OCP Layer-2 [20] approximate-timed transaction-level model as described in Section 2.2. As a result of not accounting for the bus topology and protocol overhead, the T-TLM simulation completes execution of the test program in significantly fewer cycles, giving an incomplete and inaccurate estimate of system performance, with approximately a 400% error compared to the more detailed CCATB model. The CCATB model is sensitive to changes in the bus topology and protocol, as can be seen with the varying execution performances for different topology and arbitration protocol configurations in Table 2.2, and is thus a much more suitable abstraction for exploring the communication architecture space. In the next section, we will show how CCATB models are not only faster to simulate but also to create, when compared to other abstractions currently being used for exploring the communication architecture space.
We will focus on the CCATB model for this exploration study from this point onward. In the columns for arbitration strategies, RR stands for a round robin scheme where bus bandwidth is equally distributed among all the masters. TDMA1 refers to a TDMA strategy where in every frame 4 slots are allotted to the AVLink controller, 2 slots to the USB, and 1 slot for the remaining masters. In TDMA2, 2 slots are allotted to the AVLink and USB, and 1 slot for the remaining masters. In both the TDMA schemes, if a slot is not used by a master then a secondary RR scheme is used to grant the slot to a master with a pending request. SP1 is a static priority scheme with the AVLink controller having a maximum priority followed by the USB, ARM926EJ-S, DMA, A/V Encoder and the A/V Decoder. The priorities for the AVLink controller and USB are interchanged in SP2, with the other priorities remaining the same as in SP1.

For architecture Arch1, shown in Figure 2.23 (a), performance suffers due to frequent arbitration conflicts in the shared AHB bus. The shaded cells indicate scenarios where the bandwidth constraints for the USB and/or AVLink controller are not met. From Table 2.2 we can see that none of the arbitration policies in Arch1 satisfy the constraints.

To decrease arbitration conflicts, we shift the new components to a dedicated AHB bus as shown in Figure 2.23 (b). An AHB/AHB bridge is used to interface with the main bus. We split MEM5 and attach one of the memories (MEM6) to the dedicated bus and also add an interface to the SDRAM controller ports from the new bus, so that data traffic from the new components does not load the main bus as frequently. Table 2.2 shows a performance improvement for Arch2 as arbitration conflicts are reduced. With the exception of the RR scheme, bandwidth constraints are met with all the other arbitration policies. The TDMA2 scheme outperforms TDMA1 because of the reduced load on the
main bus from the AVLink component which results in inefficient RR distribution of its 4 slots in TDMA1. TDMA2 also outperforms the SP schemes because SP schemes result in much more arbitration delay for the low priority masters (ARM CPU, DMA), whereas TDMA2 guarantees certain bandwidth even to these low priority masters in every frame.

Next, to improve performance we allocate the A/V Decoder and AVLink components to separate AHB buses, as shown in Figure 2.23 (c). From Table 2.2 we see that the performance for Arch3 improves only slightly over Arch2. The reason for the small improvement in performance is because there is not a lot of conflict (or time overlap) between transactions issued by the A/V decoder and AVLink components. As such, separating these components eliminates those few conflicts that exist between them, improving performance only slightly.

Statistics gathered during simulation indicate that the A/V decoder frequently communicates with the ARM CPU and the DMA. Therefore with the intention of improving performance even further we allocate the high bandwidth USB and AVLink controller components to separate AHB buses, and bring the A/V decoder to the main bus. Figure 2.23 (d) shows the modified architecture Arch4. Performance figures from the table indicate that the SP1 scheme performs better than the rest of the schemes. This is because the SP scheme works well when requests from the high bandwidth components are infrequent (since they have been allocated on separate buses). The TDMA schemes suffer because of several wasted slots for the USB and AVLink controller, which are inefficiently allocated by the secondary RR scheme.

We thus arrive at the Arch4 topology together with the SP1 arbitration scheme as the best choice for the new version of the SoC design. We arrived at this choice after
evaluating several other combinations of topology/arbitration schemes not shown here due to lack of space. It took us less than a day to evaluate these different communication design space points with our CCATB models and our results were verified by simulating the system with a more detailed PA-BCA model. It would have taken much longer to model and simulate the system with other approaches. The next section quantifies the gains in simulation speed and modeling effort for the CCATB modeling abstraction, when compared with other models.

2.6 Simulation and Modeling Effort Comparison

We now present a comparison of the modeling effort and simulation performance for pin accurate BCA (PA-BCA), transaction based BCA (T-BCA) and our CCATB models. For the purpose of this study we chose the SoC platform shown in Figure 2.24. This platform is similar to the one we used for exploration in the previous section but is more generic and is not restricted to the multimedia domain. It is built around the AMBA 2.0 communication architecture and has an ARM926EJ-S processor ISS model with a test program running on it which initializes different components and then regulates data flow to and from the external interfaces such as USB, switch, external memory controller (EMC) and the SDRAM controller.
For the T-BCA model we chose the approach used in the AHB CLI models [28]. Our goal was to compare not only the simulation speeds but also to determine how the speed changed with system complexity. We first compared speedup for a ‘lightweight’ system comprising of just 2 traffic generator masters along with peripherals used by these
masters, such as the RAM and the EMC. We gradually increased system complexity by adding more masters and their slave peripherals. Figure 2.25 shows the simulation speed comparison with increasing design complexity.

Note the steep drop in simulation speed when the third master was added – this is due to the detailed non-native SystemC model of the ARM926EJ-S processor which considerably slowed down simulation. In contrast, the simulation speed was not affected as much when the DMA controller was added as the fourth master. This was because the DMA controller transferred data in multiple word bursts which can be handled very efficiently by the transaction based T-BCA and CCATB models. The CCATB particularly handles burst mode simulation very effectively and consequently has the least degradation in performance out of the three models. Subsequent steps added the USB switch and another traffic generator which put considerable communication traffic and computation load on the system, resulting in a reduction in simulation speed. Overall, the CCATB abstraction level outperforms the other two models. Table 2.3 gives the average speedup of the CCATB over the PA-BCA and T-BCA models. We note that on average, CCATB is faster than T-BCA by 67% and even faster than PA-BCA models by 120%.

Table 2.3 Comparison of speedup and modeling effort

<table>
<thead>
<tr>
<th>Model Abstraction</th>
<th>Average CCATB speedup (x times)</th>
<th>Modeling Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCATB</td>
<td>1</td>
<td>~3 days</td>
</tr>
<tr>
<td>T-BCA</td>
<td>1.67</td>
<td>~4 days</td>
</tr>
<tr>
<td>PA-BCA</td>
<td>2.2</td>
<td>~1.5 wks</td>
</tr>
</tbody>
</table>
As far as modeling accuracy is concerned, CCATB models - being variants of the transaction based BCA (T-BCA) approach - are just as accurate as T-BCA models such as [28], and in some cases even more accurate, such as when compared to the T-BCA models from [25] which do not capture sufficient detail, as discussed in Section 2.2. CCATB models are also just as accurate as PA-BCA models. This is demonstrated by the fact that the execution cycle counts for the CCATB, T-BCA and PA-BCA models is the same for the case of the example we used for modeling speed and accuracy comparison in Figure 2.24. It might appear that CCATB suffers from a loss of accuracy when compared to PA-BCA models, but that is not the case, since the speedup obtained by CCATB over PA-BCA is as a result of sacrificing visibility of signals at every cycle boundary, which has a significant simulation overhead. Additionally, CCATB models use event synchronization semantics [18] to handle intra-transaction cycle level events (such as interrupts), thus capturing not only the system events occurring between different transactions, but also within transactions which can affect accuracy, and thus maintain the same level of accuracy as in PA-BCA models.

Table 2.3 also shows the time taken to model the communication architecture at the three different abstraction levels by a designer familiar with AMBA 2.0. While the time taken to capture the communication architecture and model the interfaces took just 3 days for the CCATB model, it took a day more for the transaction based BCA, primarily due to the additional modeling effort to maintain accuracy at cycle boundaries for the bus system. It took almost 1.5 weeks to capture the PA-BCA model. Synchronizing and handling the numerous signals and design verification were the major contributors for the
additional design effort in these models. CCATB models are thus faster to simulate and need less modeling effort compared to T-BCA and PA-BCA models.

**Table 2.4 Comparison of simulation time during exploration**

<table>
<thead>
<tr>
<th>Model Abstraction</th>
<th>Simulation Runs during Exploration</th>
<th>Total Simulation Time (in hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCATB</td>
<td>37</td>
<td>63.8</td>
</tr>
<tr>
<td>T-BCA</td>
<td></td>
<td>113.1</td>
</tr>
</tbody>
</table>

To understand the implication of the speedup shown in Figure 2.25, on a typical communication architecture exploration effort, we considered the multimedia subsystem shown in Figure 2.22 and performed communication architecture exploration with greater test data volume and a correspondingly much larger testbench running on the ARM processor. We performed exploration for several different combinations of topology and communication parameters. Table 2.4 shows the number of simulation runs during this exploration study, and the overall simulation time for all the simulation runs, for the CCATB and the T-BCA models. It can be seen that the exploration phase greatly benefits from the use of a faster CCATB simulation abstraction, which cuts down the simulation time by several days when compared to a similar exploration effort with the T-BCA model abstraction. With increasing communication architecture complexity, and ever increasing levels of component integration in the SoCs of the future, we believe that the CCATB modeling abstraction, which is accurate, fast to simulate and takes less time to model, will be of immense use to system architects and designers.
2.7 Conclusions

Early exploration of System-on-chip (SoC) communication architectures is extremely important to ensure efficient implementation and for meeting performance constraints. We presented the Cycle Count Accurate at Transaction Boundaries (CCATB) modeling abstraction which is a fast, efficient and flexible approach for exploring the vast communication space for shared-bus architectures in SoC designs. Our model enables plug-and-play exploration of various facets of the communication space, allowing master, slave and bus IPs to be easily replaced with their architecture variants, and quickly estimating the impact on system performance. We also proposed a five layer modeling methodology that incorporates our CCATB abstraction level in a system design flow. Interface refinement from higher abstraction levels to lower levels in the design flow is simplified as we avoid altering the interface between IPs and the communication channel as much as possible. This also eases co-simulation of SoC IPs modeled at different abstraction levels in our system flow. We described the mechanisms responsible for speedup at the CCATB modeling abstraction, which enable fast and efficient exploration of the communication design space, early in the design flow. We have successfully applied our approach for exploring several industrial strength SoC subsystems. Two such exploration case studies from the broadband communication and multimedia domains, are presented in this paper. We also showed that the CCATB models are faster to simulate than pin-accurate BCA (PA-BCA) models by as much as 120% on average and are also faster than transaction based BCA (T-BCA) models by 67% on average. In addition, the CCATB models take less time to model than T-BCA and PA-BCA models.
Chapter 3
CAPE: Early Power Estimation for On-Chip Communication Architectures using Energy Macro Models

3.1 Introduction

In addition to performance, power is another important architectural design constraint that is beginning to dominate MPSoC design, particularly for mobile applications such as cellular phones, MP3 players and laptops. These portable devices run on batteries which have a limited energy budget between charges [36]. MPSoC designs intended for use in such portable scenarios must have lower power consumption, to improve user experience. Reducing power consumption is also a priority for non-portable MPSoC applications, such as those used in server farms, which tend to consume significant amounts of power (e.g., as much as 2 megawatts for a 25,000 square foot server farm with 8000 servers [37]). With the increasing proliferation of the Internet, and mobile computing, the power problem has thus assumed a critical status that cannot be ignored by MPSoC designers.

On-chip communication architectures have a considerable impact on MPSoC power consumption. It has been reported that on-chip communication architectures can consume anywhere between 20-50% of overall system power [38] [39]. There are several reasons why the communication architecture fabric is receiving so much attention with respect to power consumption [40]. Firstly, the use of a large number of repeaters and vias to reduce wire delay in DSM technologies has almost doubled power consumption in
interconnects [41]. Secondly, state of the art communication architectures consist not only of bus wires, but also significant amounts of hardware logic (e.g., bridges, arbiters, decoders, buffers etc.) that tends to rival the amount of logic in embedded processors of moderate complexity [39]. It has been shown that communication architectures will consume a larger portion of on-chip power in future technologies [42]. Finally, the thermal implications of interconnect power consumption can no longer be ignored. A large portion of the energy consumed from the power supply in an electronic device is converted into heat. An increase of even 10 ºC in the operating temperature has been shown to not only increase interconnect delay (reducing performance), but also increase electromigration (EM), which significantly increases device failure rate [43]. Since interconnect temperatures can reach as high as 90 ºC [44], there is a need to identify potential hotspots and reduce the need for expensive cooling and packaging equipment. There is thus a dire need to create models for estimating power consumption of on-chip communication architectures as early as possible in a design flow, for better planning and design decisions.

3.1.1 Chapter Overview

In this chapter, we present models for early, system-level on-chip communication architecture power estimation. We describe a methodology to create energy macro models using multiple linear regression analysis for on-chip communication architecture power estimation, and subsequently demonstrate its utility by creating energy models for the bus matrix on-chip communication architecture [45]. The methodology makes use of detailed gate-level estimation to ensure accuracy. Once the energy models have been
created, they can be plugged into any high level cycle-accurate or cycle-approximate simulation models for power/energy estimation of MPSoC designs. This enables fast and accurate energy/power estimation for the on-chip communication architecture early in the design flow. The proposed methodology is also easily scalable across technology libraries. It takes very little time to retarget the energy models to a new technology library, while maintaining estimation accuracy.

In Section 3.2 we present an overview of the AMBA AHB bus matrix communication architecture, which we will use as a driver to demonstrate the effectiveness of our energy estimation models. In Section 3.3, we describe our energy macro model creation methodology. We also give an overview of energy macro models, and present the energy macro models for the bus matrix communication architecture. In Section 3.4, we present various experiments to demonstrate the accuracy and estimation speedup of our energy macro-models, compared to the traditional, detailed gate-level power estimation approach.

3.1.2 Related Work

There has been a lot of research activity on estimating and reducing power consumption on bus wires [46] [47] [48]. These approaches however do not consider the contribution of logic components of the communication architecture, which are now beginning to rival the level of complexity associated with moderate-sized embedded processors [39]. Recent approaches have tried to overcome this limitation by including logic components while estimating communication architecture power for hierarchical buses [39] [49] [50] [51] and network-on-chips (NoCs) [52] [53] [54] [55].
Power modeling and estimation is typically performed either at the transistor, gate or register-transfer levels [50]. In practice, most commercial design flows use register-transfer or gate-level power estimation tools. However, due to the highly complex nature of modern applications, these approaches are too inefficient for early power estimation. To overcome this drawback, recent approaches [49] [52] [53] [54] [55] [50] [51] analyze power for communication architectures at the system level, where significant improvements in run time can be achieved, at the cost of estimation accuracy. Some of these approaches extract gate-level power estimates and characterize transaction-level models [50] [51]. Although these approaches allow for fast estimation, they can be highly inaccurate and lack reusability across technology libraries. Other approaches have created energy macro-models from gate-level power estimates for the AMBA hierarchical shared bus [49], STBus interconnection network [53] and NoCs [52]. However, the accuracy of these models is not very close to gate-level estimates. In addition, none of these approaches explore system-level power estimation in the face of technology scaling. Our proposed approach enables highly accurate energy model generation, which enables fast power estimation that is easily scalable across different technology libraries.

3.2 AMBA AHB Bus Matrix Communication Architecture Overview

We use the AMBA AHB (Advanced High Performance) bus matrix [45] as a representative of bus matrix communication architectures. The AHB bus protocol supports pipelined data transfers, allowing address and data phases belonging to different transactions to overlap in time. Additional features, such as burst transfers and split
transaction support, enable high data throughputs. A bus matrix configuration consists of several AHB buses in parallel which can support concurrent high bandwidth data streams.

![Figure 3.1 AMBA AHB Full Bus Matrix](image)

Figure 3.1 shows a 2 master, 3 slave AMBA AHB full bus matrix. The **Input stage** is used to handle interrupted bursts, and to register and hold incoming transfers if receiving slaves cannot accept them immediately. The **Decoder** generates select signals for slaves, and also selects which control and read data inputs received from slaves are to be sent to the master. The **Output Stage** selects the address, control and write data to send to a slave. It calls the **Arbiter** which uses an arbitration scheme to select the master that gets to access a slave, if there are simultaneous requests from several masters. Unlike in traditional hierarchical shared bus architectures, arbitration in a bus matrix is not centralized, but distributed so that every slave has its own arbitration. Also, typically, all buses within a bus matrix have the same data bus width and frequency, which usually depends on the application characteristics.
3.3 Energy Macro-Models

In this section, we present details of our energy macro-model creation methodology. We first describe the basics of energy macro-modeling. Then we present the macro-model generation methodology. Finally, we present the energy models for all the components in the bus matrix, including bus wires.

3.3.1 Introduction to Macro Modeling

The energy consumption of a bus matrix can be obtained by identifying certain events which cause a noticeable change in energy. For this purpose, we create energy macro-models, which can capture factors that have a strong correlation to energy consumption for a given component. A macro model consists of variables, which represent factors influencing energy consumption, and regression coefficients, which capture the correlation of each of the variables with energy consumption. A general energy macro model for a component can be expressed as:

$$E_{\text{component}} = \alpha_0 + \sum_{i=1}^{n} \alpha_i \cdot \Psi_i \quad .. (3.1)$$

where $\alpha_0$ is the energy of the component which is independent of the model variables, and $\alpha_i$ is the regression coefficient for the model variable $\Psi_i$.

For the purpose of our energy macro-models, we considered three types of model variables representing factors influencing energy consumption: control, data and structural. The control factor represents control events, involving a control signal, which
triggers energy consumption either when it transitions from 1 to 0 or 0 to 1, or when it
maintains a value of 0 or 1 for a cycle. Control variables can either have a value of 1
when a control event occurs, or 0 when no event occurs, in the energy macro model
relation in Eq. (3.1). The data factor represents data events, which trigger energy
consumption on data value changes. Data variables take an integer value in Eq. (3.1)
representing the Hamming distance (number of bit-flips) of successive data inputs.
Finally, structural factors, such as data bus widths and number of components connected
to the input also affect energy consumption of a component. They are represented by
their integer values in Eq. (3.1).

![Figure 3.2 Energy Macro-model Generation Methodology](image)

**Figure 3.2 Energy Macro-model Generation Methodology**

### 3.3.2 Macro-Model Generation Methodology

A high level overview of the methodology used to create energy macro-models in this
work is shown in Figure 3.2 [114]. We start with a system testbench, consisting of
masters and slaves interconnected using the AMBA AHB bus matrix fabric. The testbench generates traffic patterns which exercise the matrix under different conditions. Synopsys Coretools [56] is used to configure the bus matrix (specify data bus width, number of masters and slaves etc.) and generates a synthesizable RTL description of the bus matrix (Step 1). This description is synthesized to the gate-level with Cadence Physically Knowledgeable Synthesis (PKS) [57], for the target standard cell library (Step 2). The gate-level netlist is then used with PrimePower [56] to generate power numbers (Step 3).

![Figure 3.3 Macro-model Template](image)

In parallel with the synthesis flow, we perform RTL simulation to generate signal waveform traces for important data and control signals (Step 4). These signal waveforms are compared with cycle energy numbers, obtained after processing PrimePower generated power report files with Perl scripts, to determine which data and control signals
in the matrix have a noticeable effect on its energy consumption. The selected data and control events become the variables in a macro-model template which consists of energy and variable values for each cycle of testbench execution (Step 5). Figure 3.3 shows an example of a macro-model template for one of the components of the bus matrix. The template consists of energy values (\textit{cycle\_energy}) and variable values (\textit{S\_load, S\_desel, HD\_addr, S\_drive}) for each cycle of testbench execution. This template is used as an input to the GNU R tool [58], which performs multiple linear regression analysis to find coefficient values for the chosen variables (Step 6). Steps 1-6 are repeated for testbenches having different structural attributes such as data bus widths and number of input components, to identify structural factors/variables that may influence cycle energy.

Statistical coefficients such as \textit{Multiple-R, R-square} and \textit{standard deviation for residuals} [59] are used to determine the goodness of fit and the strength of the correlation between the cycle energy and the model variables. Once a good fit between cycle energy and macro model variables is found, the energy macro models are generated in the final step. These models can then be plugged into any system-level cycle-accurate simulation environment, to get energy consumption values for the AMBA AHB bus matrix communication architecture.

\subsection*{3.3.3 Bus Matrix Energy Macro-Models}

To get the energy of the entire AMBA AHB bus matrix communication architecture, we used our energy macro-model generation methodology to create macro-models for each of its components. The total energy consumption of a bus matrix can be given as:
\[ E_{\text{MATRIX}} = E_{\text{INP}} + E_{\text{DEC}} + E_{\text{ARB}} + E_{\text{OUT}} + E_{\text{WIRE}} \]

where \( E_{\text{INP}} \) and \( E_{\text{DEC}} \) are the energy for the input and decoder components for all the masters in the matrix, \( E_{\text{ARB}} \) and \( E_{\text{OUT}} \) are the energy for arbiters and output stages connecting slaves to the matrix, and \( E_{\text{WIRE}} \) is the energy of all the bus wires that interconnect the masters and slaves. Energy macro-models were created for the first four factors, with \( E_{\text{WIRE}} \) being calculated separately.

The energy macro-models are essentially of the form shown in Eq. (3.1). Leakage and clock energy, which are the major sources of independent energy consumption are considered as part of the static energy coefficient \( \alpha_0 \) for each of the components. Based on our experiments, we noticed a fairly linear relationship between cycle energy and macro-model variables for the components. We will now present the energy models for each of the components.

**Input Stage:** Every master connected to a bus matrix has its own input stage, which buffers address and control bits for a transaction, if a slave is busy. The input stage model can be expressed as:

\[
E_{\text{INP}} = \alpha_{\text{inp}0} + \alpha_{\text{inp}1}.\Psi_{\text{load}} + \alpha_{\text{inp}2}.\Psi_{\text{desel}} + \alpha_{\text{inp}3}.\Psi_{\text{HDin}} + \alpha_{\text{inp}4}.\Psi_{\text{drive}}
\]

where \( \Psi_{\text{load}} \) and \( \Psi_{\text{drive}} \) are control signals asserted when the register is loaded, and when the values are driven to the slave respectively; \( \Psi_{\text{desel}} \) is the control signal from the master to deselect the input stage when no transactions are being issued; and \( \Psi_{\text{HDin}} \) is the Hamming distance of the address and control inputs to the register.
**Decoder:** A decoder component is connected to every master, and consists of logic to generate the select signal for a slave after decoding the destination address of an issued transaction. It also handles multiplexing of read data and response signals from slaves. The decoder energy consumption model can be formulated as:

\[
E_{DEC} = \alpha_{dec0} + \alpha_{dec1} \cdot \Psi_{slavesel} + \alpha_{dec2} \cdot \Psi_{respsel} + \alpha_{dec3} \cdot \Psi_{HDin} + \alpha_{dec4} \cdot \Psi_{sel}
\]

where \( \Psi_{slavesel} \) and \( \Psi_{respsel} \) are control signals asserted in the cycle in which the slave select and the data/response MUX select signals are generated, respectively; \( \Psi_{HDin} \) is the Hamming distance of the read data and response signals from the slave; and \( \Psi_{sel} \) is a control signal which transitions when the decoder is selected or deselected.

**Output Stage:** Every slave is connected to the bus matrix through the output stage, which handles multiplexing of address and control bits from the masters. It also calls the arbiter to determine when to switch between accessing masters. The energy consumption for the output stage is given by:

\[
E_{OUT} = \alpha_{out0} + \alpha_{out1} \cdot \Psi_{addrsel} + \alpha_{out2} \cdot \Psi_{dataset} + \alpha_{out3} \cdot \Psi_{HDin} + \alpha_{out4} \cdot \Psi_{noport}
\]

where \( \Psi_{addrsel} \) and \( \Psi_{dataset} \) are control signals asserted when address and data values are selected after a call to the arbiter results in a change in the master accessing the slave; \( \Psi_{HDin} \) is the Hamming distance of address and data inputs; and \( \Psi_{noport} \) is a control signal from the arbiter, which goes high when no masters access the slave in a cycle.
**Arbiter:** The arbiter is invoked by the output stage, and uses an arbitration scheme to grant access to one of the potentially several masters requesting for access to the slave. The cycle energy model for the arbiter is calculated as:

$$E_{ARB} = a_{arb0} + (a_{arb1} + n \cdot a_{arb2}) \cdot \Psi_{arb} + a_{arb3} \cdot \Psi_{arb+1} + (a_{arb4} + n \cdot a_{arb5}) \cdot \Psi_{desel} + a_{arb6} \cdot \Psi_{desel+1}$$

where $\Psi_{arb}$ and $\Psi_{arb+1}$ are control signals representing the cycle when arbitration occurs, and the subsequent cycle when the master select signal is generated; $\Psi_{desel}$ and $\Psi_{desel+1}$ are control signals representing the cycle when the arbiter is not selected by any master, and the subsequent cycle when it generates the *noport* signal for the output stage; and $n$ represents the number of masters connected to the arbiter.

**Bus Wires:** The bus wires that connect masters, slaves and logic components in the bus matrix dissipate dynamic power due to switching, and leakage power due to the repeaters inserted in long wires to reduce signal delay. The expression for energy consumption of a bus wire developed by Kretzschmar et al. [59] is extended to include the effect of repeaters (to reduce wire delay), and is given as:

$$E_{WIRE} = 0.5 V_{dd}^2 \left( \Sigma C_L + \frac{1}{d} \cdot C_{REP} + l \cdot (C_G + 2C_C) \right) \cdot \alpha + \frac{1}{d} \cdot E_{REP}$$

where $V_{dd}$ is the supply voltage, $\alpha$ is the switching factor representing bit transition activity on the wire, $\Sigma C_L$ is the sum of load capacitances of all the components connected to the wire, including the driver and receiver, $C_{REP}$ is the capacitance of a repeater, $C_G$ is the wire-to-ground capacitance per unit length, $C_C$ is the coupling capacitance per unit
length of the wire to its adjacent wires, \( l \) is the length of the wire, \( d \) is the inter-repeater distance and \( E_{REP} \) is the repeater internal energy.

This single bus wire model is extended for an \( N \) bit bus. The Berkeley Predictive Technology Model (PTM) [60] is used to estimate values for ground \( (C_G) \) and coupling \( (C_C) \) capacitances. The static energy of the repeater \( (E_{REP}) \) and its capacitance \( (C_{REP}) \) are obtained from data sheets. The component load capacitance on the wire \( (C_L) \) is obtained after component synthesis. Repeater capacitance and static energy is obtained from data sheets. A high-level simulated-annealing based floorplanner (PARQUET) [61] is used to generate IP block placement to obtain wire lengths and an approach proposed by Cong et al. [62] is used to determine optimal-delay repeater spacing/sizing. Finally, the switching factor \((\alpha)\) is obtained from simulation.

### Table 3.1 Coefficients for Energy Macro-Models (180nm)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Stage</td>
<td>( \alpha_{inp0} )</td>
<td>7.78</td>
<td>( \alpha_{inp3} )</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td>( \alpha_{inp1} )</td>
<td>3.81</td>
<td>( \alpha_{inp4} )</td>
<td>3.27</td>
</tr>
<tr>
<td></td>
<td>( \alpha_{inp2} )</td>
<td>2.60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decoder</td>
<td>( \alpha_{dec0} )</td>
<td>0.47</td>
<td>( \alpha_{dec3} )</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>( \alpha_{dec1} )</td>
<td>3.04</td>
<td>( \alpha_{dec4} )</td>
<td>0.38</td>
</tr>
<tr>
<td></td>
<td>( \alpha_{dec2} )</td>
<td>2.17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Stage</td>
<td>( \alpha_{out0} )</td>
<td>0.72</td>
<td>( \alpha_{out3} )</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>( \alpha_{out1} )</td>
<td>2.61</td>
<td>( \alpha_{out4} )</td>
<td>1.48</td>
</tr>
<tr>
<td></td>
<td>( \alpha_{out2} )</td>
<td>1.53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arbiter</td>
<td>( \alpha_{arb0} )</td>
<td>0.65</td>
<td>( \alpha_{arb4} )</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td>( \alpha_{arb1} )</td>
<td>0.76</td>
<td>( \alpha_{arb5} )</td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td>( \alpha_{arb2} )</td>
<td>0.30</td>
<td>( \alpha_{arb6} )</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>( \alpha_{arb3} )</td>
<td>0.60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.4 Experiments

3.4.1 Energy Macro-model Generation

In order to generate the energy macro-models for the AHB bus matrix communication architecture, we first selected a diverse set of bus matrix based system testbenches, having different number of master and slave components, bus widths, and data traffic profiles that activate the components in the bus matrix in different states. We used these to generate the energy macro-models and obtained the component regression coefficients using the methodology from Figure 3.3, for the TSMC 180nm standard cell library. The regression coefficients for the macro-models of each of the components of the bus matrix (Section 3.3) are shown in Table 3.1. Also shown are the R squared (R²) values which measure the goodness of fit for each regression [108]. The value of R² is a fraction between 0.0 and 1.0. A value of 0.0 indicates that knowing the variable values will not allow us to predict the energy values, whereas a value of 1.0 indicates that knowing the variable values will allow us to predict the value of energy perfectly. From the table it can be seen that the R² values are close to 1.0, thus enabling reliable prediction of energy at the system-level.

<table>
<thead>
<tr>
<th>Standard cell library</th>
<th>$a_{inp0}$</th>
<th>$a_{inp1}$</th>
<th>$a_{inp2}$</th>
<th>$a_{inp3}$</th>
<th>$a_{inp4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>180 nm</td>
<td>7.78</td>
<td>3.81</td>
<td>2.60</td>
<td>0.96</td>
<td>3.27</td>
</tr>
<tr>
<td>130 nm</td>
<td>1.33</td>
<td>0.66</td>
<td>0.04</td>
<td>0.24</td>
<td>0.56</td>
</tr>
<tr>
<td>90 nm</td>
<td>1.23</td>
<td>0.44</td>
<td>0.02</td>
<td>0.20</td>
<td>0.40</td>
</tr>
<tr>
<td>65 nm</td>
<td>0.54</td>
<td>0.29</td>
<td>0.02</td>
<td>0.09</td>
<td>0.18</td>
</tr>
</tbody>
</table>
Next, we targeted the same system testbenches to the 130nm, 90nm and 65nm TSMC standard cell libraries and repeated the regression coefficient generation process for the components in the bus matrix. The regression coefficients for the Input Stage bus matrix component, for each of the standard cell libraries, are shown in Table 3.2. It can be seen from the table that as we reduce the standard cell size, the coefficient values decrease, which is indicative of a decrease in overall power consumption. However, note that the reduction in coefficient values is not linear and the table indicates a change in relative importance of model variables with a change in standard cell library. For instance, the value of the coefficient for model variable $\alpha_{inp3}$ is less than $1/3^{rd}$ the value of the coefficient for $\alpha_{inp4}$ at 180nm, but this ratio reduces to $1/2$ for the 65nm library.

A major advantage of our energy macro-model approach is the ease with which it can be retargeted to different standard cell libraries. Typically retargeting to a new standard cell library is a very time-intensive effort. However, our approach simply requires the generation of gate-level cycle energy numbers with the new library for the small system testbenches; these cycle energy numbers are used to update the cycle energy column in the macro-model template shown in Figure. 3.4, after which the regression analysis is repeated. Once the gate level cycle energy numbers are available for the standard cell library, it only takes a few minutes to obtain the new coefficients for each of the components in the bus matrix. This enables rapid retargeting of models for new cell libraries and results in an order-of-magnitude reduction in development time.
3.4.2 Accuracy of Energy Macro-model

We performed experiments to determine the macro-model accuracy for different technology libraries, by comparing our macro-model energy estimates with detailed gate-level energy estimates. We selected 4 system testbenches that were separate and distinct from the ones used for characterization and generation of the energy macro models, and had different bus matrix structures and traffic characteristics: (i) a 2 master, 3 slave bus matrix with 32 bit data bus width (2x3_32b), (ii) a 3 master, 4 slave bus matrix with 32 bit data bus width (3x4_32b), (iii) a 4 master, 5 slave bus matrix with 32 bit data bus width (4x5_32b) and (iv) a 2 master, 3 slave bus matrix with 64 bit data width (2x3_64b). We synthesized these architectures and estimated cycle energy values using PrimePower [56] at the gate-level, for each of the 180, 130, 90nm and 65nm TSMC standard cell libraries. In parallel, we characterized the model variables and coefficient values in the energy macro-models for each of the components in the matrix, to obtain estimated energy values.

![Figure 3.4 Average cycle energy estimation errors](image-url)
Figure 3.4 compares the average error in energy estimated at each cycle by the macro-models compared to gate-level estimation, for the different standard cell libraries. It can be seen that the energy macro-models allow highly accurate cycle energy estimation which is unaffected by the scaling of technology libraries towards deep sub-micron (DSM). *The maximum average estimation error for the macro-models is only 4.19%.* The variation in error for different test cases is due to the approximation introduced by curve fitting during the regression phase. This error can be further reduced if more variables (events) are included in the energy macro model characterization phase.

Next, we plugged the energy macro-models into a transaction-level bus cycle-accurate (T-BCA) simulation environment in SystemC [63] [18]. The T-BCA simulation abstraction captures the bus matrix at a cycle-accurate granularity and uses function/transaction calls instead of signals to obtain simulation speed over bus-cycle accurate (BCA) models which capture signal details, without sacrificing accuracy [17]. The simulation model incorporates bus matrix energy macro-models for cycle energy estimation. The equations from Section 3.3 are inserted in the code for each component, at points where a change in the value of an energy consuming event (i.e. dependent variable) can occur.

Figure 3.5 shows a snapshot of the power waveform generated by gate-level simulation using PrimePower, and the SystemC T-BCA simulation using our energy macro-models for the 2x3_32b bus matrix system testbench. As can be seen, the power estimates using the system-level T-BCA simulation model are highly correlated to the actual power consumption. This high correlation highlights *an additional benefit of the methodology in estimating peak energy* (as opposed to average energy). Peak energy is
important in the planning of power grids which is an important but becoming increasingly difficult to do in DSM.

![Graph showing predicted and measured power waveforms](image)

**Figure 3.5 Predicted and measured power waveforms**

<table>
<thead>
<tr>
<th>Test bench</th>
<th>PrimePower Gate-level CPU time</th>
<th>SystemC T-BCA level CPU time</th>
<th>Speedup (X times)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x3_32b</td>
<td>2.58 hrs</td>
<td>9.1 sec</td>
<td>1021</td>
</tr>
<tr>
<td>3x4_32b</td>
<td>7.91 hrs</td>
<td>18.8 sec</td>
<td>1515</td>
</tr>
<tr>
<td>4x5_32b</td>
<td>27.04 hrs</td>
<td>49 sec</td>
<td>1987</td>
</tr>
<tr>
<td>2x3_64b</td>
<td>3.10 hrs</td>
<td>9.9 sec</td>
<td>1127</td>
</tr>
</tbody>
</table>

Table 3.3 Comparing time taken for power estimation

Table 3.3 compares the CPU time taken for the PrimePower simulation (for gate-level cycle energy estimation) with the system-level T-BCA based prediction, for the four system testbenches described earlier. The time taken for the gate-level and system-level power estimation does not depend on the technology library used, and is independent of it. As can be seen from the table, the system-level power estimation gives a substantial speedup close to **2000X** over gate-level power estimation using PrimePower. Note that in
our comparison we have not included time taken for value change dump (VCD) file generation and parasitic RC extraction file generation needed for PrimePower based estimation, which takes from several minutes to hours for each of the testbenches. From these speedup results and our earlier observation on the high estimation accuracy of the macro-models across technology libraries, we believe that our approach can be very useful for designers interested in fast and accurate communication architecture power estimation, early in the design flow, at the system level.

3.5 Conclusion

In this chapter, we presented an energy macro-model based power estimation methodology that can quickly and accurately estimate power for different technology libraries at the system-level. Energy macro-models were created for the bus matrix on-chip communication architecture using our proposed methodology, and the models were shown to have an average cycle energy estimation error of less than 5% across the 180, 130, 90 and 65nm technology libraries, compared with gate-level estimation. Plugging these macro-models in a system-level simulation framework allows a substantial speedup of almost 2000X for cycle energy estimation, over gate-level estimation. Note that the accuracy of these models can be further improved at the cost of development time and model complexity. For instance, a quadratic or higher order macro-model may achieve greater estimation accuracy, but such a model might take longer to develop, since finding the appropriate variables and determining exact higher order dependencies requires a non-trivial effort from the designer. We believe that our general approach can be ported to any on-chip (custom or standard) communication architecture and be immensely useful
for designers interested in fast and accurate on-chip communication architecture power estimation and exploration, early in the design flow at the system level.
Chapter 4  
Automated Multi-Constraint On-Chip Communication Architecture Synthesis

4.1 Introduction

The design of on-chip communication architectures is becoming harder and harder as the number of components integrated into a single embedded multi-processor system-on-chip (MPSoC) increases due to greater market demands for convergence devices. The increasing number of components in systems translates into more inter-component communication that must be handled by the on-chip communication infrastructure. The pull of multi-dimensional design constraints such as performance, power, cost, area and time-to-market further complicate the communication architecture design process. Designs are expected to support high performance and reliability, along with lower power consumption, cost, area and time-to-market. These design goals often conflict with each other – for instance, creating a communication architecture configuration that supports high performance data transfers by adding more buses and increasing bus clock frequencies ends up increasing power consumption, cost and area. The design of bus-based communication architectures thus requires techniques to not only optimize for individual design goals, but also focus on making design decisions that can provide a good balance between various design goals such as power, performance, cost and area.

The bus-based communication architecture design space is a union of two sub-spaces: (i) the topology space, which is concerned with the number of buses, how the number of
buses are inter-connected and the allocation on components on these buses, and (iii) the parameter space, which is concerned with the values of bus architecture parameters such as arbitration schemes, bus widths, bus clock frequencies, out-of-order (OO) buffer sizes and DMA burst sizes. A manual traversal of this enormous design space has now become inefficient and practically infeasible even for small-sized systems that can have millions of combinations of possible bus architecture configurations. Designers need to partially, or better still, completely automate the process of exploring and designing communication architectures that satisfy application constraints. The synthesis of bus-based communication architectures must therefore address this need for automated synthesis, while exploring the vast design space in an efficient manner.

4.1.1 Chapter Overview

In this chapter, we address the issues highlighted above by proposing an automated on-chip communication architecture synthesis framework to perform application-specific, system-level trade-offs between multiple constraints – power, performance and cost. Our synthesis framework [114] [117] uses several optimization techniques to prune the massive communication design space and generates a set of eligible configurations which are used to generate trade-off reports for the designer. We make use of our novel CCATB simulation abstraction (described in Chapter 2) for fast and accurate communication architecture performance estimation. We also make use of our communication architecture energy macro models (described in Chapter 3) to quickly and accurately estimate power/energy consumption of the on-chip communication architecture fabric.
While our synthesis framework is applicable to various commonly used bus architectures such as hierarchical shared buses and bus matrix (or crossbar bus), we will demonstrate our synthesis framework in the context of the bus matrix on-chip communication architecture. The reason for selecting the bus matrix communication architecture is that these architectures are an evolution of the hierarchical shared buses, and are increasingly being adopted in designs today. They consist of multiple parallel buses that can provide superior bandwidth response, while keeping the simple, standard interface of bus-based communication architectures, which makes them suitable candidates as communication architectures for MPSoC designs today and in the near future. In addition, while there are several different bus standards that support the bus matrix configuration, we will focus on the AMBA AHB bus matrix [45] standard while describing our techniques in this chapter. However, the framework is applicable other bus standards such as STBus [30] and SONICS [31], that support the matrix architecture.

In Section 4.2, we briefly present the AMBA AHB bus matrix, and discuss the goals of synthesis for the bus matrix communication architecture. In Section 4.3, we present some background and then elaborate on our synthesis framework. In Section 4.4, we present experiments on several industrial strength applications from the networking domain to show the effectiveness of our framework.

4.1.2 Related Work

There is a significant body of research in the area of bus architecture synthesis. Early work was aimed at minimizing bus width [64], interface synthesis and simple synchronization protocol selection [65] and topology generation for simple buses without
arbitration [66]. Ryu et al. [67] performed studies to find optimal bus topologies for a SoC design. Pinto et al. [68] proposed an algorithm for constraint-driven topology synthesis under the assumption that relative positions of components were fixed. Murali et al. [69] proposed an automated synthesis approach for STBus crossbars. However, their work primarily deals with automated crossbar topology synthesis – the communication parameters (arbitration schemes, OO buffer sizes, bus widths and speeds) which have considerable influence on system performance are not explored or synthesized. Additionally, Murali et al. [69] assume that critical data streams cannot overlap on the same bus, places a static limit on the maximum number of components that can be attached to a bus, and also require the designer to specify hard-to-determine threshold values of traffic overlap as an input, based on which components are allocated to separate buses. These are conservative approaches which lead to an overdesigned, sub-optimal system. Lyonnard et al. [70] proposed a synthesis flow which supported shared bus and point to point connection templates. These templates have to be parameterized manually, which makes the process time consuming. Lahiri et al. [71] designed communication architectures after exploring different solutions using fast performance simulation. However, they assumed the bus topology to be given. Shin et al. [72] used a genetic algorithm for automating the generation of bus architecture parameters to meet performance requirements. However, they do not focus on bus topology synthesis. Our framework differs from these existing approaches in the way we automate the synthesis of not only the bus topology, but also the generation of values for bus architecture parameters.
As far as the target design constraints during synthesis are concerned, there have been several approaches proposed for dealing with performance-oriented system-level synthesis of hierarchical shared bus [66] [73] [74] [75] [109], NoCs [76] and, more recently, bus matrix/crossbar [69] [77] architectures. Some approaches also consider power-aware synthesis for hierarchical shared buses [48] and NoCs [78], but power-aware synthesis for bus matrix architectures has not been addressed. There have also been a few pieces of work that have looked at power-performance tradeoffs for segmented buses [79] and NoCs [80]. Our framework differs from existing work in that we focus on enabling trade-offs between multiple design constraints such as power, performance and cost during on-chip communication architecture synthesis.

4.2 Full and Partial AMBA AHB Bus Matrix

We use the AMBA AHB (Advanced High Performance) bus matrix [5] as a representative of bus matrix communication architectures, and use it as a driver for our synthesis framework.

![Figure 4.1 AMBA AHB Full Bus Matrix](image-url)
Figure 4.1 shows a 2 master, 4 slave AMBA AHB full bus matrix. The *Input stage* is used to handle interrupted bursts, and to register and hold incoming transfers if receiving slaves cannot accept them immediately. The *Decoder* generates select signals for slaves, and also selects which control and read data inputs received from slaves are to be sent to the master. The *Output Stage* selects the address, control and write data to send to a slave. It calls the *Arbiter* that uses an arbitration scheme to select the master that gets to access a slave, if there are simultaneous requests from several masters. Unlike in traditional hierarchical shared bus architectures, arbitration in a bus matrix is not centralized, but distributed so that every slave has its own arbitration. Also, typically, all buses within a bus matrix have the same data bus width, which usually depends on the application.

One drawback of the *full bus matrix* structure shown in Figure 4.1 is that it connects every master to every slave in the system, resulting in a large number of wires and logic components in the matrix. While this configuration ensures high bandwidth for data transfers, it is certainly not optimal as far as power consumption is concerned. Figure 4.2 shows a *partial bus matrix* that has fewer wires and components (e.g., arbiters, MUXs),
which consequently reduces power consumption, as well as area, at the cost of performance (due to an increase in the number of shared links). Points A, B and C in Figure 4.2 are referred to as slave access points (SAPs). Our synthesis framework (described in the next section) starts with a full bus matrix, and aims to generate a set of partial bus matrix configurations, with different number of buses and logic components that meet all the performance constraints of the application being considered. The system designer can then trade-off different design goals such as power, performance and cost on the generated set of candidate solutions.

4.3 Synthesis Framework

In this section, we describe our automated synthesis framework for fast system-level, application-specific, on-chip communication architecture synthesis that enables trade-offs between multiple design constraints. First we describe how performance constraints are represented in our approach. Next, we present our assumptions and goals. Finally, we describe the flow for the framework in detail.

4.3.1 Background

Typically, MPSoC designs have performance constraints that are dependent on the nature of the application, and must be satisfied by the underlying on-chip communication architecture. The *throughput* of communication between components is a good measure of the performance of a system [66]. To represent performance constraints in our approach, we define a **Communication Throughput Graph** $CTG = G(V,A)$ that is a
directed graph, where each vertex $v$ represents a component in the system, and an edge $a$ connects components that need to communicate with each other. A **Throughput Constraint Path** (TCP) is a sub-graph of a CTG, consisting of a single component for which data throughput must be maintained, and other masters, slaves and memories that are in the critical path impacting the maintenance of the throughput. Figure 4.3 shows a CTG for a SoC subsystem, with a TCP involving the ARM2, MEM2, DMA and ‘Network I/F’ components, where data packets must stream out of ‘Network I/F’ at a rate of at least 1 Gbps.

![Figure 4.3 Communication Throughput Graph (CTG)](image)

**4.3.2 Assumptions and Goals**

We are given an MPSoC application which has certain minimum performance constraints that need to be satisfied. The application has several components (IPs) that need to communicate with each other. We assume that hardware-software partitioning has taken place and that the appropriate functionality has been mapped onto hardware and software IPs. These IPs are standard “black box” library components that cannot be
modified during the synthesis process. The target bus matrix communication architecture standard (e.g., AMBA AHB bus matrix) is also specified.

The goal of our bus matrix synthesis framework then, is to automatically generate a set of bus matrix configurations for the given application that meet the minimum performance constraints of the application. The output of the framework is a set of power, performance and cost trade-off graphs for the generated set of valid bus matrix solutions, allowing a designer to pick a solution with the desired combination of characteristics.

4.3.3 Simulation Environment

Since communication behavior in a system is characterized by unpredictability due to dynamic bus requests from cores, contention for shared resources, buffer overflows etc., a simulation based approach is necessary for accurate performance estimation. However, relying solely on simulation based exploration can limit the amount of space that can be explored in a reasonable amount of time. Our framework makes use of a combination of static and simulation based dynamic analysis to speed up the synthesis process.

For the simulation part of our flow, we capture behavioral models of components and bus architectures in SystemC and keep them in an IP library database. Since we were concerned about the speed of simulation, we use the our CCATB modeling abstraction (described in Chapter 2) which enables a fast transaction-based, bus cycle accurate simulation, with average simulation speeds of 150–200 Kcycles/sec, while running embedded software applications on processor ISS models. The communication model in this abstraction is extremely detailed, capturing delays arising due to frequency and data
width adapters, bridge overheads, interface buffering and all the static and dynamic delays associated with the standard bus architecture protocol being used.

4.3.4 Communication Parameter Constraint Set

The exploration space for a typical MPSoC bus matrix communication architecture consists of combinations of bus topology configurations with communication parameter values for bus clock speeds, OO buffer sizes and arbitration schemes. If we allow these parameters to have any arbitrary values, an incredibly vast design space is created. The time required to traverse this space as we search for the most cost effective configuration (which also satisfies all performance constraints) would become unreasonably large. More importantly, once we manage to find such a system configuration, there would be no guarantee that the values generated for the communication parameters would be practically feasible. To ensure that our framework generates a realistic bus matrix communication architecture configuration, we allow the designer to specify a Communication Parameter Constraint Set (Ψ). The constraints are in the form of a discrete set of valid values for the communication parameters to be synthesized. We allow the specification of two types of constraint sets for components – a global constraint set (Ψ_G) and a local constraint set (Ψ_L). The designer can specify local constraints for a resource if these constraint values are different from the global constraints. Otherwise, the designer can leave the local constraints unspecified, thus allowing the resource to inherit the more generic global constraints. For instance, a designer might set the allowable bus clock speeds for a set of buses in a subsystem to multiples of 33 MHz, with a maximum speed of 166 MHz, based on the operation
frequency of the cores in the subsystem, while globally, the allowed bus clock speeds are multiples of 50 MHz, up to maximum of 400 MHz. The presence of a local constraint overrides the global constraint, while the absence of it results in the resource inheriting global constraints. This provides a convenient mechanism for the designer to bias the synthesis process based on knowledge of the design and the technology being targeted. Such knowledge about the design is not a prerequisite for using our synthesis framework. As long as $\Psi$ is populated with any discrete set of values for the parameters, our framework will attempt to synthesize a feasible, low-cost optimal matrix architecture. However, informed decisions can greatly reduce the time taken for synthesis and help the designer generate a more practical system.

4.3.5 Synthesis Flow

We now describe our bus matrix communication architecture synthesis framework in more detail. The basic idea is to start with a full bus matrix configuration, and iteratively cluster slaves together to reduce the logic components and wires in the matrix (i.e. cost) that will intuitively allow a reduction in power consumption. But this will come at the cost of performance, which degrades with decreasing number of buses, because of bottlenecks at the slave end due to increased traffic (and consequently extended arbitration delays). The final solution set will consist of bus matrix configurations having different number of components and buses (i.e., costs), without violating application performance constraints, lying between the extremes of a full bus matrix and an optimally reduced, partial bus matrix having the least number of buses and components possible. The trade-off graphs for the solution set will then allow a designer to select a
configuration having the desired power, performance and cost characteristics for a particular application.

Figure 4.4 Bus Matrix Synthesis Framework Flow

Figure 4.4 shows the major steps in the synthesis framework. The inputs are (i) the application-specific Communication Throughput Graph (CTG), (ii) a library of IP components consisting of masters, slaves and memories, (iii) a template of the target full bus matrix communication architecture (e.g., AMBA AHB bus matrix), (iv) energy macro-models for the bus matrix, and (v) a communication parameter constraint set ($\Psi$) – which includes $\Psi_G$ and $\Psi_L$, that allow a designer to set the bus matrix bus clock.
frequency, data bus width (assumed to be uniform for all buses in the matrix) and arbitration schemes. The output is a set of solutions that meet application performance constraints, and a set of power-performance-cost-area trade-off reports for these solutions.

We start by first mapping components in the IP library onto the full bus matrix template (Phase 1). The full matrix configuration is then simulated at the TLM level, with no arbitration contention overhead since there are no shared channels and also because we assume infinite ports at IP interfaces. We also set the OO buffer sizes to the maximum allowed in $\Psi$. The TLM simulation allows us to obtain application-specific data traffic statistics such as number of transactions on a bus, average transaction burst size on a bus and memory usage profiles. Knowing the bandwidth to be maintained on a channel from the Throughput Constraint Paths (TCPs) in the CTG, we can also estimate the minimum clock speed at which any bus in the matrix must operate, in order to meet its throughput constraint, as follows. The data throughput ($\Gamma_{TLM/B}$) from the TLM simulation, for any bus $B$ in the matrix is given by

$$\Gamma_{TLM/B} = \frac{(numT_B \times sizeT_B \times widthB \times \Omega_B)}{\sigma}$$

where $numT$ is the number of data transactions on the bus, $sizeT$ is the average size of these data transactions, $width$ is the data bus width, $\Omega$ is the clock speed, and $\sigma$ is the total number of cycles of TLM simulation for the application. The values of $numT$, $sizeT$ and $\sigma$ are obtained from the TLM simulation in Phase 1. To meet the throughput constraint $\Gamma_{TCP/B}$ for the bus $B$,
\[
\Gamma_{\text{TLM}/B} \geq \Gamma_{\text{TCP}/B}
\]

\[
\therefore \quad \Omega_B \geq (\sigma \times \Gamma_{\text{TCP}/B}) / (\text{width}_B \times \text{num}_B \times \text{size}_B)
\]

The minimum bus clock speed thus found is used to create (or update) the local bus speed constraint set \(\Psi_{L\text{(speed)}}\) for bus \(B\).

In the next phase (Phase 2), we perform preprocessing on this matrix structure by (i) removing unused buses with no data transfers on them, according to the CTG, and (ii) migrating components that are accessed exclusively by a master, to its local bus, in order to reduce components (arbiters, output stage) and wire congestion in the matrix.

Next, we perform static analysis to further reduce the number of components and buses in the matrix, by using a *branch and bound based hierarchical clustering algorithm* to cluster slave components (Phase 3). Note that we do not consider clustering masters in the matrix, in our approach. While clustering masters can result in some savings for simple SoC systems, for the highly parallel, high performance multi-processor SoC applications that we target, clustering masters can drastically degrade system performance. This is because master clustering adds two levels of contention, one at the master end and another at the slave end, in a data path, which lengthens the completion time for transactions issued by any of the clustered masters. Additionally, clustering masters also severely limits the parallelism in the system, since if one master in a ‘master cluster’ is active with a transaction, for instance a burst of several data transfers, none of the other masters in that cluster can issue transactions. In our experience, even increasing
the bus clock frequency to compensate for the reduced parallelism and longer transaction latency in the system does not prevent throughput constraint violations when masters are clustered.

Before describing the algorithm, we present a few definitions. A slave cluster \( SC = \{s_1, ..., s_n\} \) refers to an aggregation of slaves that share a common arbiter. Let \( M_{SC} \) refer to the set of masters connected to a slave cluster \( SC \). Next, let \( \Pi_{SC1/SC2} \) be a superset of sets of buses which are merged when slave clusters \( SC1 \) and \( SC2 \) are merged. Finally, for a merged bus set \( \beta = \{b_1, ..., b_n\} \), where \( \beta \subset \Pi_{SC1/SC2} \), let \( K_\beta \) refer to the set of allowed bus speeds for the newly created bus when the buses in set \( \beta \) are merged, and is given by

\[
K_\beta = \Psi_L(speed)(b_1) \cap \Psi_L(speed)(b_2) \cap ... \cap \Psi_L(speed)(b_n)
\]

The branching algorithm starts out by clustering two slave clusters at a time, and evaluating the gain from this operation. Initially, each slave cluster has just one slave. The total number of clustering configurations possible for a bus matrix with \( n \) slaves is given by \( \binom{n}{2} + \binom{n}{2} \cdot \binom{n}{2} + \binom{n}{2} \cdot \binom{n}{2} \cdot \binom{n}{2} + ... + \frac{(n! \times (n-1)!)}{2^{(n-1)}} \). This creates an extremely large exploration space, which cannot be traversed in a reasonable amount of time. In order to consider only valid clustering configurations, we make use of a bounding function.

Figure 4.5 shows the pseudo code for our bounding function which is called after every clustering operation of any two slave clusters \( SC1 \) and \( SC2 \). In Step 1, we use a look up table to see if the clustering operation has already been considered previously, and if so, we discard the duplicate clustering. Otherwise we update the lookup table with
the entry for the new clustering. In Step 2, we check to see if the clustering of SC1 and SC2 results in the merging of buses in the matrix, otherwise the clustering is not beneficial and the solution can be bounded. If the clustering results in bus mergers, we calculate the number of merged buses for the clustering and store the cumulative weight of the clustering operation in the branch solution node. In Step 3, we check to see if the allowed set of bus speeds for every merged bus is compatible or not. If the allowed speeds for any of the buses being merged are incompatible (i.e., $K_\beta = \emptyset$ for any $\beta$), the clustering is not possible and we bound the solution. Additionally, we also calculate if the throughput requirement of each of the merged buses can be theoretically supported by the new merged channel. If this is not the case, we bound the solution. The bounding function thus enables a conservative pruning process which quickly eliminates invalid solutions and allows us to rapidly converge on the optimal solution.

<table>
<thead>
<tr>
<th>Step 1:</th>
<th>if (exists lookupTable(SC1,SC2)) then discard duplicate clustering else update lookupTable(SC1, SC2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2:</td>
<td>if ($M_{SC1} \cap M_{SC2} == \emptyset$) then bound clustering else $cum_{weight} = cum_{weight} +</td>
</tr>
<tr>
<td>Step 3:</td>
<td>for each set $\beta \in \Pi_{SC1/SC2}$ do if ($K_\beta == \emptyset$)</td>
</tr>
</tbody>
</table>

Figure 4.5 bound function
Figure 4.6 shows the branch and bound clustering flow for the example shown earlier in Figures 4.1, 4.2 and 4.3. Every valid branch in the solution space corresponds to a valid clustering of slave components and is represented by a node in the figure. The nodes annotated with an X correspond to clustering solutions that are eliminated by the bounding function in Figure 4.5 for being duplicate solutions; nodes annotated with a B correspond to solutions that do not meet the other criteria in the bounding function. The figures above the nodes correspond to the cumulative weights of the clustering solution. This allows us to determine the quality of the solution – the node with the maximum cumulative weight corresponds to a bus matrix with the least number of buses. The
highlighted node in Figure 4.6 corresponds to the optimal solution, with the lowest cost while still satisfying performance constraints.

The output of the clustering algorithm is a set of solutions that are ranked (according to cost) and stored in a database according to the number of buses. The next phase (Phase 4) selects a solution from the ranked database, and does the following: (i) fixes arbitration schemes for each slave cluster in the matrix, and (ii) performs simulation, to gather power and performance statistics. The arbitration scheme gives priority to higher *Throughput Constraint Paths* (TCPs) from the CTG. Simulation is performed using our transaction-level CCATB [12] simulation model in SystemC, as discussed earlier. It allows us to verify if all performance (TCP) constraints are satisfied, in the next step (Phase 5), since it is possible for dynamic factors such as traffic congestion, and buffer overflows etc. to invalidate the statically predicted solution, in some cases. In order to obtain power/energy statistics for the application, we plug in our bus matrix energy macro-models (presented in Chapter 3) into the simulation model. The macro-model equations from Chapter 3 (Section 3.3.3) are inserted in the code, at points where a change in the value of an energy consuming event (i.e., dependent variable) can occur. To illustrate the linking of a transaction in the CCATB model with the energy macro-models, we present an example of a single write transaction and show how it activates the macro-models as it propagates through the bus matrix. Table 4.1 shows the dependent variables from Chapter 3 (Section 3.3.3), for each of the components in the matrix that are triggered during the write transaction in the model, assuming no slave delays and a single cycle overhead for arbitration. Every change in a dependent variable for a component triggers its corresponding energy macro-model equation, which calculates the energy for the event.
and updates the cycle energy value. Static energy values are updated at the end of every cycle. Energy values can be recorded at each cycle, for every component if needed; or in a cumulative manner for the entire bus matrix, for use in the final power-performance trade-off analysis.

Table 4.1 Dependent Variable Activations for a Write Transaction

<table>
<thead>
<tr>
<th>Component</th>
<th>Cycle n</th>
<th>Cycle n+1</th>
<th>Cycle n+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Stage</td>
<td>(\Psi_{load}, \Psi_{H Din})</td>
<td>(\Psi_{drive})</td>
<td>-</td>
</tr>
<tr>
<td>Decoder</td>
<td>(\Psi_{slavesel}, \Psi_{H Din})</td>
<td>(\Psi_{respsel})</td>
<td>-</td>
</tr>
<tr>
<td>Arbiter</td>
<td>(\Psi_{arb})</td>
<td>(\Psi_{arb+1})</td>
<td>-</td>
</tr>
<tr>
<td>Output Stage</td>
<td>-</td>
<td>(\Psi_{addrsel}, \Psi_{H Din})</td>
<td>(\Psi_{datares}, \Psi_{H Din})</td>
</tr>
</tbody>
</table>

Phases 4 and 5 are repeated for the desired number of times, based on the number of solutions required in the output set. The designer can specify different strategies to select solutions from the ranked database, such as selecting solutions from the top and the bottom of the rankings, and then selecting solutions at regular intervals between them, to get a wider spread on the power-performance characteristics. Finally, we use the energy and performance statistics obtained from simulation in Phase 4, for each of the solutions in the output set (having different areas and costs), to generate power-performance-cost-area trade-off graphs.

4.4 Experiments

In this section, we present two sets of experiments to show the effectiveness of our automated synthesis framework. The first set of experiments compares the quality of the
solutions generated by our framework with existing work, for industrial strength MPSoC designs. The second set of experiments uses our synthesis framework to generate power-performance-cost-area trade-offs for an industrial strength MPSoC design, similar to the one used in the first set of experiments.

4.4.1 Quality of Synthesis Framework

To compare the quality of the synthesis results using our framework, we chose the closest existing piece of work that deals with automated bus matrix synthesis by Murali et al. [69]. In their work, the goal is to generate a low cost bus matrix (i.e., having the minimum number of buses) which satisfies performance constraints. To compare our framework with their approach, we select an industrial strength MPSoC application from the networking domain and apply our synthesis framework, as well as the framework developed by Murali et al. [69]. We will first describe the MPSoC application, followed by the comparison of results.

Figure 4.7 shows the CTG for the SIRIUS industrial MPSoC application from the networking domain, used for data packet processing and forwarding. For clarity, the TCPs are presented separately in Table 4.2. ARM1 is a protocol processor (PP) while ARM2 and ARM3 are network processors (NP). The ARM1 PP is responsible for setting up and closing network connections, converting data from one protocol type to another, generating data frames for signaling, operating and maintenance and exchanging data with NP using shared memory. The ARM2 and ARM3 NPs directly interact with the network ports and are used for assembling incoming packets into frames for the network connections, network port packet/cell flow control, assembling incoming packets/cells
into frames, segmenting outgoing frames into packets/cells, keeping track of errors and gathering statistics. The ASIC1 block performs hardware cryptography acceleration for DES, 3DES and AES. The DMA is used to handle fast memory to memory and network interface data transfers, freeing up the processors for more useful work. Besides these master cores, SIRIUS also has a number of memory blocks, network interfaces and peripherals such as interrupt controllers (ITC1, ITC2), timers (Watchdog, Timer1, Timer2), UART and a packet accelerator (Acc1).

Figure 4.7 CTG for SIRIUS MPSoC application
Table 4.2 Throughput Constraint Paths (TCPs) For SIRIUS

<table>
<thead>
<tr>
<th>IP cores in Throughput Constraint Path (TCP)</th>
<th>Throughput Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM1, MEM1, DMA, SDRAM1</td>
<td>640 Mbps</td>
</tr>
<tr>
<td>ARM1, MEM2, MEM6, DMA, Network I/F2</td>
<td>480 Mbps</td>
</tr>
<tr>
<td>ARM2, Network I/F1, MEM3</td>
<td>5.2 Gbps</td>
</tr>
<tr>
<td>ARM2, MEM4, DMA, Network I/F3</td>
<td>1.4 Gbps</td>
</tr>
<tr>
<td>ASIC1, ARM3, SDRAM1, Acc1, MEM5, Network I/F2</td>
<td>240 Mbps</td>
</tr>
<tr>
<td>ARM3, DMA, Network I/F3, MEM5</td>
<td>2.8 Gbps</td>
</tr>
</tbody>
</table>

Table 4.3 Customizable Parameter Constraint Set (SIRIUS)

<table>
<thead>
<tr>
<th>Set</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus speed</td>
<td>25, 50, 100, 200, 300, 400</td>
</tr>
<tr>
<td>arbitration strategy</td>
<td>static, RR, TDMA/RR</td>
</tr>
<tr>
<td>OO buffer size</td>
<td>1 – 8</td>
</tr>
</tbody>
</table>

Table 4.3 shows the global customizable parameter set $\Psi_G$. For the synthesis we target an AMBA 3 AXI based bus matrix communication architecture, with the goal of generating an optimal, minimal cost architecture, for comparison with the approach from Murali et al. [69]. Figure 4.8 shows the optimal matrix structure output by our synthesis flow, which satisfies all six throughput constraints in the design (Table 4.2). The data bus width used in the matrix is 32 bits, and the slave-side arbitration strategies, operating speeds for the buses and OO buffer sizes (for components supporting OO transaction completion) are shown in the figure. While the full bus matrix architecture used 95 buses, the final synthesized matrix further reduces the number of buses to as few as 16 (this includes the local buses for the masters) which is almost a $6 \times$ saving in the number of buses used when compared to the original full bus matrix. The entire synthesis process took just a few hours to complete instead of the several days or even weeks it would have taken for a manual effort.
Next, we applied the methodology proposed by Murali et al. [69] to the same MPSoC application, with the aim of obtaining a minimal cost bus matrix. Since their approach only generates matrix topology (while we generate both topology and parameter values), we restricted our comparison to the number of buses in the final synthesized design. The mixed integer linear programming (MILP) threshold based approach proposed by Murali et al. [69] requires the designer to statically specify (i) the maximum number of slaves per cluster and (ii) the traffic overlap threshold, which if exceeded prevents two slaves from being assigned to the same bus cluster. The results of our comparison study are shown in Figure 4.9. BMSYN is our bus matrix synthesis approach while the other
comparison points are obtained from the approach proposed by Murali et al. [69]. \( S(x) \), for \( x = 0, 10, 20, 30, 40 \), represents the threshold based approach where no two slaves having a traffic overlap of greater than \( x\% \) can be assigned to the same bus, and the X-axis in Figure 4.9 varies the maximum number of slaves allowed in a bus cluster for these comparison points. Note that the values of \( x \) are chosen based on the recommendations in the approach from Murali et al. [69]. It is clear from the figure that our synthesis approach produces a lower cost system (having lesser number of buses) than approaches which force the designer to statically approximate application characteristics.

![Figure 4.9 Comparison with threshold based approach for SIRIUS](image)

**Figure 4.9** Comparison with threshold based approach for SIRIUS

<table>
<thead>
<tr>
<th>Applications</th>
<th>Processors</th>
<th>Masters</th>
<th>Slaves</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIPER</td>
<td>2</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>SIRIUS</td>
<td>3</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>ORION4</td>
<td>4</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>HNET8</td>
<td>8</td>
<td>13</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 4.4 Number of Cores in MPSOC Applications
Next, in order to show the quality of our synthesis results, we applied our framework on three other industrial strength MPSoC applications. Table 4.4 shows details of these applications, along with SIRIUS. While VIPER and SIRIUS are variants of existing industrial strength applications, ORION4 and HNET8 are larger systems which have been derived from the next generation of MPSoC applications currently in development. The table shows the number of components in each of these applications. The Masters column includes the processors in the design, which are primarily ARM based microprocessors. Our goal from the synthesis effort was to obtain the lowest cost solution, which would still satisfy the performance constrains specified by the designer, for each of the applications.

![Figure 4.10 Comparison of number of buses for MPSoC applications](image)

Figure 4.10 shows a comparison between the number of buses in a full bus matrix, and the number of buses in the optimal, low cost solution generated by our synthesis framework. It can be seen that there is a substantial reduction in the number of buses (cost) of the bus matrix after synthesis with our framework. The savings for the MPSoC
applications range 4.6× to 9× when compared with a full bus matrix, which can significantly lower the cost of the MPSoC design, and gives an indication of the quality of optimization possible with our framework.

4.4.2 Trade-off Generation with Synthesis Framework

In the previous section, we saw how our comprehensive synthesis framework produces superior quality results, compared to existing work in literature. In this section, we illustrate how our synthesis framework enables trade-offs between multiple design constraints during synthesis. For this purpose, we applied our synthesis framework on an industrial MPSoC application, similar to SIRIUS above. Figure 4.11 shows the Communication Throughput Graph (CTG) for the application, with its minimum performance constraints that need to be satisfied, represented by Throughput Constraint Paths (TCPs) presented separately in Table 4.5.

ARM1 is a protocol processor (PP), while ARM2 and ARM3 are network processors (NP). The ARM1 PP is mainly responsible for setting up and closing network connections, converting data from one protocol type to another and generating data frames for signaling, operating and maintenance. The ARM2 and ARM3 NPs directly interact with the network ports and are used for assembling incoming packets into frames for the network connections, network port packet/cell flow control, assembling incoming packets/cells into frames, segmenting outgoing frames into packets/cells, keeping track of errors and gathering statistics. ARM4 is used for specialized data processing involving data encryption. The DMA is used to handle fast memory to memory and network
interface data transfers, freeing up processors for more useful work. Besides these master cores, the application also has a number of memory blocks, network interfaces and peripherals such as interrupt controllers (ITC1, ITC2), timers (Watchdog, Timer1, Timer2), UARTs (UART1, UART2) and data packet accelerators (ASIC1, ASIC2).

Figure 4.11 CTG for networking MPSoC application

Table 4.5 Throughput Constraint Paths (TCPs)

<table>
<thead>
<tr>
<th>IP cores in Throughput Constraint Path (TCP)</th>
<th>Throughput Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM1, MEM1, DMA, MEM3, MEM5</td>
<td>320 Mbps</td>
</tr>
<tr>
<td>ARM1, MEM3, MEM4, DMA, Network I/F2</td>
<td>240 Mbps</td>
</tr>
<tr>
<td>ARM2, Network I/F1, MEM2</td>
<td>800 Mbps</td>
</tr>
<tr>
<td>ARM2, MEM6, DMA, MEM8, Network I/F2</td>
<td>200 Mbps</td>
</tr>
<tr>
<td>ARM3, ARM4 , Network I/F3, MEM2, MEM7</td>
<td>480 Mbps</td>
</tr>
</tbody>
</table>
The application and the target AMBA AHB bus matrix architecture were captured at the CCATB abstraction in SystemC, with the bus matrix energy macro-models plugged into the simulation environment. The bus matrix design constraints specified a data bus width of 32 bits, a clock frequency of 100 MHz and a static priority-based arbitration scheme. Power numbers are calculated for the TSMC 0.18µm standard cell library. We also included the energy contribution of bus wires as described in Section 3.3.3 of Chapter 3.

![Figure 4.12 Power-Performance-Cost Trade-off graph](image1)

**Figure 4.12 Power-Performance-Cost Trade-off graph**

![Figure 4.13 Component-wise Power Comparison](image2)

**Figure 4.13 Component-wise Power Comparison**
Figure 4.12 shows the power-performance curves for the different cost solutions output by the framework for the MPSoC application. The $x$-axis shows solutions in the output set having different number of buses (i.e. costs), while the $y$-axis shows the % change in power and performance, using the original full bus matrix as the base case. It can be seen that reducing the number of buses reduces the average power dissipation, because of a smaller number of arbiters, output stages and bus wires in the matrix, which results in less static and dynamic power consumption. Figure 4.13 highlights this trend, showing a comparison of the component-wise power consumption for the full bus matrix and the solution having the least number of buses, which also consumes the least power. While the power consumed by the input stage initially decreases when the number of buses is decreased, due to reduced port switching, this trend is soon offset as traffic congestion increases arbitration delays, requiring additional buffering of transactions, for solutions with lesser number of buses. Similarly, for bus wires, the initial power reduction due to reduced number of wires in the matrix is soon offset by the need for longer shared wires to connect the increasingly clustered slaves.

![Figure 4.14 Total Energy vs. Runtime Trade-off graph](image)

**Figure 4.14 Total Energy vs. Runtime Trade-off graph**
As far as the performance change is concerned, (measured in terms of average % change in data throughput of constraint paths) a reduction in the number of buses increases traffic congestion and reduces performance due to an increase in arbitration wait time. However it is interesting to note that there are certain points (e.g., point A) in Figure 4.12 where reducing the number of buses actually improves performance! This happens because of a reduction in port switching at the master end as slave clusters grow, reducing re-arbitration delays for masters. In addition to the average % change in throughput, another useful metric to gauge application performance is its total runtime. Figure 4.14 shows the corresponding total energy vs. application runtime trade-off graph for the different cost solutions of the MPSoC that is useful for cases where the application must execute within a given time or energy budget, such as for mobile battery-driven applications.

![bus matrix power vs. chip area](image)

**Figure 4.15 Power vs. Chip-Area Trade-off graph for MPSoC**
It is also possible to perform other design space trade-offs within the CAPPS framework. For instance, Figure 4.15 shows the pareto-optimal trade-off curve between bus matrix power consumed and the total area of the chip. To obtain this curve, we iterated through several different floorplans using a high level simulated annealing based floorplanner [88], which we integrated into our automated synthesis framework. Since different floorplans have different wire routing arrangements, every floorplan has a different value for power consumption of the bus wires. Typically, as the allowed chip area is increased, the floorplanner can optimize the floorplan better, by reducing bus wire length, which consequently reduces bus power consumption.

Overall, the power-performance curves show a possible trade-off of up to approximately 20% for power and up to 40% for performance, enabling a designer to select the appropriate point in the solution space which meets the desired power and performance characteristics. There is also a possible trade-off of up to 10% for runtime and up to 15% for total energy consumed, which is a useful trade-off metric when the application is running on a fixed energy budget, which is the case for mobile battery-driven devices. The automated system-level synthesis framework generated the solution set and statistics in a matter of a few hours, instead of days or even weeks it would have taken with a gate-level estimation flow.

4.5 Conclusion

In this chapter, we presented an automated on-chip communication architecture synthesis framework that enables fast system-level, application-specific, power-performance-cost-area trade-offs in bus matrix communication architecture synthesis.
Our synthesis framework comprehensively generates both the topology and protocol parameters (such as bus speeds, OO buffer sizes and arbitration strategies) for an on-chip communication architecture. We presented experimental results on industrial strength MPSoC applications that showed how our framework produces superior quality results, compared to existing work in literature. We also presented experiments to show how our framework can generate comprehensive trade-offs, in a matter of a few hours, between different combinations design constraints such as power, performance, cost, energy, runtime and area. Such a framework is invaluable for designers early in the design flow, for quick and reliable communication architecture design space exploration, to guide design decisions at the system level.
Chapter 5
FABSYN: Floorplan-Aware On-Chip Communication Architecture Synthesis

5.1 Introduction

Typically, on-chip communication architecture synthesis approaches create an on-chip communication architecture without considering physical implementation issues such as the layout of the components on the chip or the lengths and routing of the bus wires interconnecting the components. Such physical-level information can be extremely important to factor into a synthesis effort, in order to guarantee that the synthesis results are reliable, especially in DSM technologies. However, such physical level information is typically available much later in the design flow, and therefore it can be challenging to abstract up this information to early in the design flow, during communication architecture design. A workaround is to make use of high level floorplanners to give estimates of component placement on the chip, and then using routing estimates to obtain the layout and length of the interconnection wires between components. For IP-based design, the area and form factor of the IP components is typically assumed to be known, for components that are being reused from earlier designs, or approximated, using various techniques. This allows early block placement and floorplan generation of designs, to allow consideration of physical-level issues during on-chip communication architecture synthesis.
5.1.1 Chapter Overview

In this chapter, we describe an automated, floorplan-aware on-chip communication architecture synthesis approach that considers physical implementation issues while designing the communication architecture. We present the floorplan-aware methodology in the context of hierarchical shared bus type on-chip communication architecture synthesis effort. The primary goal of our synthesis framework is to synthesize a low cost hierarchical bus on-chip communication architecture while satisfying all performance constraints of the application. In addition, we incorporate physical implementation awareness into the synthesis process, by making use of a high level floorplanner and wire length (and delay) estimates, early in the design flow. This allows us to detect possible bus clock timing violations and eliminate them early in the design flow. Typically, such timing violations are otherwise detected much later in the design flow, at the physical planning and routing stages, where it takes much longer, and requires greater effort to fix the problem. Note that although we present our floorplan-aware methodology in the context of a hierarchical shared bus on-chip communication architecture synthesis effort in this chapter, the physical awareness steps can be easily appended to the bus matrix on-chip communication architecture synthesis flow that was presented in the last chapter.

In Section 5.2, we present our assumptions and the problem description for our synthesis framework. In Section 5.3, we describe our floorplanning and wire delay estimation engines, which introduce physical implementation awareness into our synthesis framework. In Section 5.4, we describe our automated physically-aware hierarchical bus-based on-chip communication architecture synthesis framework in
detail. Finally, in Section 5.5, we present experiments to demonstrate the effectiveness of our proposed approach, with several industrial strength MPSoC applications.

5.1.2 Related Work

There have been a few approaches in literature that have made use of a high level floorplanner to introduce physical awareness during on-chip communication architecture design. Early work by Drinic et al. [81] proposed using a high level floorplanner to create a block placement, and estimate global wiring delays to ensure that hard real-time communication deadlines between components were satisfied during bus topology synthesis. A genetic algorithm was used to iterate over different bus topology configurations having low contention (and task assignments on components), with wire delay information from the floorplanner guiding the bus topology creation process. Dick et al. [81] and Meguerdichian et al. [82] used a high-level floorplanner to determine feasibility during bus topology synthesis, by comparing estimates of wire length with an upper bound on wire length. However an upper bound on wire length has the disadvantage of not accounting for varying capacitive loads of the components. A similar approach is used by Thepayasuwan et al. [83] [84] [85] to obtain wire lengths during bus topology synthesis. However, instead of comparing wire lengths against a statically determined upper bound as is done by Dick et al. [81] and Meguerdichian et al. [82], the lengths are incorporated into a simulated annealing cost function that is used to synthesize the bus topology. Simulated annealing subsequently minimizes the cost function, and selects a topology solution with low total wire length. Hu et al. [86] used a high level floorplanner to determine wire lengths during a point-to-point interconnect
width parameter synthesis, with the aim of obtaining wire energy. Guo et al. [87] used a floorplanner during segmented bus topology synthesis. The floorplanner aims to reduce the length of critical wires with high switching activity, in order to reduce wire energy consumption. Unlike these existing approaches, we make use of a high level floorplanner and wire delay estimation heuristics in order to identify bus cycle time violations and verify the feasibility of the synthesized bus architecture early in the design flow. We believe that this step will become increasingly important in the deep submicron era as clock speeds increase and lengthy propagation delays cause violations of timing constraints that will need to be detected and corrected early in the design flow.

5.2 Assumptions and Problem Formulation

We are given an application for which we assume the HW/SW partitioning has already been performed. The resulting SoC design has possibly several hardware and software components (IPs) onto which application functionality has been mapped and which need to communicate with each other. The standard bus-based communication architecture (e.g., AMBA, CoreConnect) which determines the pins at the IP interface and for which the bus topology and communication parameter values must be synthesized, is also specified. The IPs are assumed to be standard “black box” library components which cannot be modified during the bus synthesis process, except for the memory components.

Typically, SoC designs need to satisfy performance requirements that are dependent on the nature of the application. The throughput of communication between components is a good measure of the performance of a system. We assume that we are given one or more throughput constraints for the system that need to be satisfied. These constraints can
involve communication between two or more IPs. Figure 5.1 shows a **Communication Throughput Graph (CTG)** which is a directed graph, where each vertex $v$ represents a component in the system, and an edge $a_{ij}$ connects components $i$ and $j$ that need to communicate with each other. Each vertex $v$ contains information about the component it represents, such as its area, dimensions (fixed width/height or bounds on aspect ratio), capacitive loads on output pins and which bus type it can be connected to – a *main* high bandwidth bus like AHB, a *peripheral* low bandwidth bus like APB, or both. An edge $a_{ij}$ is associated with a throughput constraint $\tau(a_{ij})$ if it lies within a **throughput constraint path (TCP)**. Figure 5.1 shows a TCP involving CPU1, MEM1, S1 and M2 components, where the rate of data packets streaming out of M2 must not fall below 360 Mbps. A TCP, in general, has a single master for which data throughput must be maintained and other masters, slaves and memories which are in the critical path that impacts the maintenance of the throughput.

![Figure 5.1 Communication Throughput Graph (CTG)](image)
The goal of our communication architecture synthesis framework is to determine the number of buses and the allocation of SoC IPs on these buses (bus topology synthesis), and generate values for arbitration priorities, data bus widths, bus clock speeds and DMA burst sizes (bus architecture parameter synthesis) for the selected standard bus-based communication architecture, while ensuring that all system throughput constraints are satisfied. In addition, we want to consider layout information of the chip to detect bus cycle time violations early in the design flow, so that we can modify the bus architecture to eliminate these violations which might otherwise take up costly design iterations later in the flow. This leads us to our problem definition:

**Problem Definition** A bus \( B \) can be considered to be a partition of the set of components \( V \) in a CTG, where \( B \subset V \). Then our primary objective is to determine a component to bus assignment for a hierarchical bus architecture, such that the partitioning of \( V \) onto \( N \) buses results in a minimal number of buses \( |N| \) and satisfies bus cycle timing constraints, while meeting all performance requirements in the design, represented by the TCPs in a CTG. As a secondary objective, we attempt to reduce the clock speeds and data widths of the buses in the synthesized solution.

### 5.3 Floorplanning and Wire Delay Estimation Engines

The floorplanning stage in a typical design flow arranges arbitrarily shaped, but usually rectangular blocks representing circuit partitions, into a non-overlapping placement while minimizing a cost function, which is usually some linear combination of die area and total wirelength. Our floorplanning engine is adapted from the simulated
annealing-based floorplanner proposed in [88]. The input to the floorplanner is a list of components and their interconnections in the system. Each component has an area associated with it (obtained from RTL synthesis). Dimensions in the form of width and height (for “hard” components) or aspect ratio (for “soft” components) are also required for each component. Additionally, maximum die size and fixed locations for hard macros can also be specified as inputs. Given these inputs, our floorplanner minimizes the cost function

\[
Cost = w_1 \cdot Area + w_2 \cdot Bus_{WL} + w_3 \cdot Total_{WL} \quad \ldots (5.1)
\]

where \(Area\) is the area of the chip, \(Bus_{WL}\) is the wire length corresponding to wires connecting components on a bus, \(Total_{WL}\) is total wire length for all connections on the chip (including inter-bus connections) and \(w_1, w_2, w_3\) are adjustable weights which are used to bias the solution. The floorplanner outputs a non-overlapping placement of components from which the wire lengths can be calculated by using half-perimeter of the minimum bounding box containing all terminals of a wire (HPWL) [89].

\[\sum_{i=1}^{k} C_i - C_L \quad \quad C_L = \sum_{j=1}^{k} \frac{\sum_{i=1}^{j} l_i}{l_j} \cdot C_j\]

**Figure 5.2 Transforming multiple pin net into a two pin net**
Once the wire lengths have been calculated, the delay estimation engine is invoked. The wire delay is calculated based on formulations proposed in [90]. The inputs to this stage are the wire lengths from the floorplanner and the capacitive loads (C\textsubscript{L}) of component output pins (obtained from RTL synthesis). We can simplify the multiple pin problem (which is representative of a bus line) depicted in Figure 2(a) to a two pin problem shown in Figure 2(b). Then the delay for a wire of length \( l \), with optimal wire sizing (OWS) [21], is given as

\[
T = R_d C_o + \left( \frac{\alpha_1 l}{W^2(\alpha_2 l)} + \frac{2 \alpha_1 l}{W(\alpha_2 l)} + R_d c_f + \sqrt{R_d r c_o l} \right) l
\]

... (5.2)

where \( \alpha_1 = \frac{1}{4} r c_o \), \( \alpha_2 = \frac{1}{2} \sqrt{\frac{r c_o}{R_d C_L}} \) and \( W(x) \) is Lambert’s \( W \) function defined as the value of \( w \) which satisfies \( w e^w = x \). \( R_d \) is the resistance of the driver, \( l \) is the wire length, \( C_o \) and \( C_L \) are capacitive loads which are calculated as shown in Figure 2(c) and the rest of the parameters are dependent on the process technology used – \( r \) is the sheet resistance in \( \Omega/\text{sq} \), \( c_o \) is unit area capacitance in \( \text{fF}/\mu\text{m}^2 \) and \( c_f \) is unit fringing capacitance in \( \text{fF}/\mu\text{m} \) (defined to be the sum of fringing and coupling capacitances). The values for these technology dependent parameters are listed in Table 5.1, and have been calculated using the SIA roadmap [91].
Table 5.1 Parameters based on NTRS 97

<table>
<thead>
<tr>
<th>Tech (µm)</th>
<th>0.18</th>
<th>0.15</th>
<th>0.13</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>0.068</td>
<td>0.073</td>
<td>0.081</td>
</tr>
<tr>
<td>( c_a )</td>
<td>0.060</td>
<td>0.054</td>
<td>0.046</td>
</tr>
<tr>
<td>( c_f )</td>
<td>0.064</td>
<td>0.054</td>
<td>0.043</td>
</tr>
</tbody>
</table>

Figure 5.3 Example floorplan

The delay estimation engine is ultimately used to check for bus cycle time violations in the design. This is illustrated through an example. Figure 5.3 shows a floorplan for a system where IP1 and IP2 are connected to the same bus as ASIC1, Mem4, ARM, VIC and DMA, and the bus has a speed of 333 Mhz. This implies that the bus cycle time is 3 ns. For a 0.13 µm process and a driver resistance value \( R_d \) of 0.4 kΩ, the floorplanner finds a wire length of 9.9 mm between pins connecting the two IPs to the bus, with \( C_L = 2.936 \text{ pF} \) and \( C_O = 0.988 \text{ pF} \) for the wire. The wire delay, obtained by inserting these
values in (5.2), is found to be 3.5 ns, which clearly violates the clock cycle time constraint of 3 ns. In this way our floorplanning and wire delay estimation engines can determine if a synthesized design has buses with clock cycle timing violations. Typically, once such violations are detected at the physical implementation stage in the design flow, designers end up pipelining the buses by inserting latches, flip-flops or register slices on the bus, in order to meet bus cycle time constraints. However, we found that such pipelining of the bus can not only have an adverse effect on critical path performance, but also requires tedious manual reworking of RTL code and extensive re-verification of the design, which can be very time consuming. As we will show later, our synthesis flow attempts to automatically eliminate such violations once they are detected, early in the design flow at the system level.

5.4 Synthesis Framework

In this section, we describe our automated, physically-aware hierarchical shared bus architecture synthesis framework. First, we will present a few definitions that will be used later when explaining the synthesis flow in more detail.

**Definitions:** Let $CTG = G(V_{CTG}, A_{CTG})$ be a Communication Throughput Graph, where $V_{CTG}$ is the set of vertices, each of which represents a component (a master or a slave) in the design, and $A_{CTG}$ is the set of edges used to connect the components in $V_{CTG}$ that need to communicate which each other. $SLV_{CTG}$ is the set of slave components in $V_{CTG}$, where $SLV_{CTG} \subseteq V_{CTG}$. $MEM_{CTG}$ is the set of memory components in $SLV_{CTG}$, such that $MEM_{CTG} \subseteq SLV_{CTG}$. $L_{CTG} = \{l_1, l_2, ..., l_n\}$ is a set of slave leaf components (i.e., slave components with a single incident edge connecting them to a single master component).
in the CTG ($L_{CTG} = SLV_{CTG}$), and where $\text{master}(l_n)$ refers to the master connected to the leaf component $l_n$. Next, let $\Omega = \{TCP_1, TCP_2, ..., TCP_n\}$ be a superset of all throughput constraint paths (TCPs) in a CTG, where each TCP in $\Omega$ is itself a set of vertices representing the components that are part of the TCP, as discussed previously in Section III-B. $MAST_{TCP_n}$ is the set of master components and $SLV_{TCP_n}$ is the set of slave components in the constraint path $TCP_n$, such that $TCP_n = SLV_{TCP_n} \cup MAST_{TCP_n}$.

![Figure 5.4 Hierarchical bus architecture synthesis flow](image)

Figure 5.4 Hierarchical bus architecture synthesis flow

We now describe our automated synthesis framework in detail. Figure 5.4 gives a high level overview of the flow. The inputs to the flow include a Communication Throughput Graph, a target bus-based communication architecture standard (e.g., AMBA), a set of Communication Parameter Constraints ($\Psi$) and a library of behavioral IP models. The
general idea is to first perform preprocessing transformations on the CTG to improve the performance of the entire system (preprocess) and then map all the components from the CTG to a simple bus topology of the target bus-based communication architecture. Then, we iteratively select a Throughput Constraint Path (TCP) from set $\Omega$, starting from the TCP with the most stringent constraint, and search the communication parameter space for a suitable parameter configuration (explore_params) and possibly perform topology mutations if needed (mutate_topology) till the TCP constraint is satisfied. Once all TCP constraints are satisfied, we optimize the design (optimize_design) to further lower the cost of the system. Next we invoke the floorplanning and delay estimation engines to detect bus cycle time violations. If timing violations are detected, we update $\Omega$ with the TCPs having components on the buses with violations, and use a feedback loop to re-enter the flow to repeat the topology mutation and parameter exploration phase to eliminate these violations or proceed to output the synthesized system and floorplan once there are no violations.

---

**Step 1:** map CTG to communication protocol-independent TLM  
**Step 2:** simulate design; record memory data traffic profiles  
**Step 3:**  
for each mem_node $\in$ MEM$_{CTG}$  
if NonOverlapAccess(mem_node)  
SplitNode(mem_node)  
**Step 4:**  
for each slave_node $\in$ L$_{CTG}$  
MergeNode(slave_node, master(slave_node))

---

**Figure 5.5 preprocess procedure**

Figure 5.5 shows the pseudo code for the preprocess stage. In the first step we map the components in the CTG from the behavioral IP library database to a bus protocol-
independent, transaction-level simulation model in SystemC having a virtual channel for every edge in the graph. This model has no contention since there are no shared channels and also because we assume infinite ports at IP interfaces. The purpose of this step is to obtain, through simulation, a memory usage profile (Step 2). Once we have obtained this profile, we attempt to split those memory nodes for which different masters access non-overlapping regions (Step 3). Finally we merge local slave nodes with their master nodes to reduce contention and loading on shared buses (Step 4). Note that we perform Step 3 before Step 4 because it allows us to generate local memories which can then be merged with their corresponding masters.

Step 1: Set bus speed, bus width to maximum allowed in set $Ψ$
Step 2: Select a combination of valid arbitration priority ordering and valid DMA burst size. Exit if all valid combinations exhausted
Step 3: Simulate design
Update best result configuration
Step 4: If TCP constraint not satisfied or previously satisfied TCP constraint violated, goto Step 2
Step 5: Update $Ω$ and arbitration priority ordering for masters in TCP
Step 6: Simulate design for remaining DMA burst sizes
Update allowed DMA burst size set.

**Figure 5.6 explore_params procedure**

After the preprocess stage, all the components in the enhanced CTG and the selected bus architecture are mapped from the IP library database to our fast transaction-based CCATB simulation model (Chapter 2) with a simple bus topology – a single shared main and a single shared peripheral bus. As mentioned earlier, every node in a CTG has information relating to the type of bus it can be connected to, which guides the mapping process. A bus $B$ can be considered to be a partition of nodes $V_{CTG}$ in a CTG, such that $B$
\( \subset V_{CTG} \). Once the simple topology has been created, we select the largest unsatisfied TCP constraint from set \( \Omega \) and search for a suitable combination of communication parameter values to satisfy the constraint in the \textit{explore\_params} stage (Figure 5.4). Figure 5.6 gives the pseudo code for this procedure. \textit{explore\_params} searches for a suitable combination of parameter values which satisfies the TCP constraint under consideration, for the current bus topology. The parameter values are bounded by the constraint set \( \Psi \) specified by the designer. However, the exploration space arising from the combinations of the bounded values can still be very large. In the interest of achieving practical running times, we must further prune this space.

We start by decoupling the bus widths and speeds from the arbitration schemes and DMA burst sizes. We set the bus widths and speeds to the maximum allowed values set by the designer in \( \Psi \) (Step 1). We do this because if TCP constraints are not met for the maximum values of bus widths and speeds, they will certainly not be met for lower values of these parameters. We cannot, however, set the DMA burst size to its maximum value and the arbitration priority to a fixed value, and make the same guarantee. Therefore Step 1 allows us to quickly prune only the bus width and speed parameter space. Next, we select a combination of a valid arbitration priority ordering and DMA burst size, and then proceed to simulate the design (Steps 2, 3). The best result configuration in Step 3 is the combination of parameters for which the least number of TCP constraints are violated and the throughput for the TCP being considered is the highest. The set of valid arbitration priorities is governed by the following rules: (a) priorities of masters in TCPs with larger throughput constraints are always greater than priorities of masters in TCPs with lower throughput constraints, (b) once a TCP
constraint is satisfied, the relative arbitration priority ordering for masters in the TCP is updated (Step 5) and not changed anymore and (c) only combinations of priority orderings within the TCP under consideration need to be explored if the previous two rules are followed. These three rules reduce the large arbitration space and make it more manageable. The set of valid DMA burst sizes is governed by the following rule: (a) once a TCP constraint is satisfied, only those DMA burst size values which did not violate the satisfied TCP constraint are considered for subsequent TCPs. Thus, as TCP constraints are satisfied, the set of valid DMA burst size values shrinks, reducing the DMA burst size exploration space. Figure 5.6 shows how once a TCP constraint is satisfied, we simulate the design for different DMA burst size values to generate an updated set of allowed DMA burst sizes (Step 6) which will be used for subsequent TCP explorations.

If the TCP constraint is not satisfied for any combination of communication parameter values, we attempt to change the communication topology in the mutate_topology stage. Figure 4.7 shows the pseudo code for this procedure. To meet TCP constraints, we need to eliminate conflict on shared buses, and this can be done by creating a new bus and migrating IPs, from the TCP being considered, iteratively to the new bus till the conflict is resolved.

In mutate_topology, we first check to see if this is the first time that the procedure has been called, and if so, then we create a new bus, choose an unselected master at random and migrate the master to the new bus (Steps 2, 6). If it is the first time that the procedure has been called, then none of the masters in MAST_TCP have been previously selected for migration, and the function call NoneSelected(MAST_TCP) returns a true value. In subsequent invocations of mutate_topology, we iteratively migrate the slaves in SLV_TCP to
the new bus (Steps 3, 7). The function call $\text{AllSelected}(SLV_{TCP})$ returns a false value if there are any remaining slaves in $SLV_{TCP}$ which have yet to be selected for migration. Once all slaves in $SLV_{TCP}$ have been considered for migration and the TCP is still not satisfied, we check for unselected masters in the current TCP (Step 4). If there are still unselected masters remaining, we undo all slave migrations since the last master migration by calling $\text{UndoNodeMigration}(SLV_{TCP})$, mark the slaves as being unselected and migrate a randomly chosen previously unselected master to the new bus (Steps 4, 6).

---

**Step 1:** if previous mutation violates a satisfied TCP constraint,  
\[\text{UndoPreviousNodeMigration}()\]

**Step 2:** if (NoneSelected(MAST_{TCP}))  
\[B = \text{CreateNewBus}()\]
\[\text{Goto Step 6}\]

**Step 3:** if (!$\text{AllSelected}(SLV_{TCP})$)  
\[\text{Goto Step 7}\]

**Step 4:** if (!$\text{AllSelected}(MAST_{TCP})$) && ($\text{AllSelected}(SLV_{TCP})$)  
\[\text{UndoNodeMigration}(SLV_{TCP})\]
\[\text{MarkUnselected}(SLV_{TCP})\]
\[\text{Goto Step 6}\]

**Step 5:** if ($\text{AllSelected}(MAST_{TCP})$) && ($\text{AllSelected}(SLV_{TCP})$)  
\[\text{MarkUnselected}(SLV_{TCP})\]
\[\text{MarkUnselected}(MAST_{TCP})\]
\[m\_node = \text{random}(\text{unselected}(MAST_{TCP}))\]
\[\text{MarkSelectedPermanent}(m\_node)\]
\[B = \text{CreateNewBus}()\]
\[\text{if (!$\text{AllSelected}(MAST_{TCP})$)  
\[\text{Goto Step 6}\]
\[\text{else}\]
\[\text{Goto Step 6}\]

**Step 6:** $m\_node = \text{random}(\text{unselected}(MAST_{TCP}))$  
\[\text{MarkSelected}(m\_node)\]
\[\text{MigrateNode}(m\_node, B)\]
\[\text{exit};\]

**Step 7:** $s\_node = \text{random}(\text{unselected}(SLV_{TCP}))$  
\[\text{MarkSelected}(s\_node)\]
\[\text{MigrateNode}(s\_node, B)\]
\[\text{exit};\]

---

**Figure 5.7 mutateTopology procedure**
In subsequent invocations of `mutate_topology`, we again migrate the slaves to the new bus (Steps 3, 7). After all masters and slaves in the current TCP have been moved to the new bus or at least considered for migration, it is possible that the TCP constraint is still not met (Step 5). In that case, we mark all the master and slaves in the TCP as unselected, randomly select a master on the previously created bus and permanently assign it to that bus, create another bus and starting from a randomly selected master (or a randomly selected slave if there are no more masters to migrate), we iteratively migrate IPs to that bus (Steps 5, 6). In this way, new buses are created till enough bandwidth is available to satisfy the TCP constraint. Note that if a topology mutation causes the best result configuration from `explore_params` to violate any previously satisfied TCP constraints, we undo the mutation (Step 1). Otherwise we keep the mutation, even if it deteriorates current TCP performance slightly. This allows us to take into account the effect of local minima in the exploration phase.

---

**Step 1:** Select previously unselected bus from generated bus architecture
**Step 2:** Reduce data bus width to next lower value
Simulate design
**Step 3:** If (TCP constraint violation), undo, else goto step 2
**Step 4:** Reduce bus speed to next lower value
Simulate design
**Step 5:** If (TCP constraint violation), undo, else goto step 4
**Step 6:** If all buses examined, exit, else goto step 1

---

**Figure 5.8 optimize_design procedure**

Once all the TCP constraints are satisfied, we arrive at the `optimize_design` stage. The pseudo code for this stage is shown in Figure 5.8. The purpose of this stage is to reduce
the maximum values we selected earlier for bus widths and bus clock speeds. Here we iteratively consider each bus in the system and attempt to lower the value for data bus width (Step 2) and bus clock speed (Step 4), without violating any TCP constraints. Reducing the bus width reduces the number of wires in a bus and lowers cost of the system. Reducing the bus speed on the other hand reduces the probability of a bus cycle time violations since it lengthens the bus clock cycle time period. The order in which the bus width or the bus speed is reduced is flexible, and is left to the designer.

Next we pass the optimized system through our floorplanning and wire delay estimator engine. If a timing violation is detected (as discussed in Section 5.3), the set $\Omega$ is updated with TCPs which have components on the buses with violations, and we use a feedback loop to go back and attempt to eliminate these violations. Since the cumulative capacitive load of components directly contributes to increasing signal propagation delay (Section 5.3), we attempt to reduce the number of components on the bus having a violation. Therefore, when we go back into the flow using the feedback loop, we first select the TCP from $\Omega$ which has components on the violated bus with the largest load capacitance on its pins, and iteratively migrate them to another existing bus, (or a new bus if migration to existing buses causes TCP constraint violations). If there is still a violation, we select another TCP from $\Omega$ and migrate components from that TCP away from the violated bus. We also give higher priority to reducing bus clock speed over reducing data bus width in the optimize_design stage, since reducing bus clock speed improves the probability of meeting the bus clock cycle period constraint. Note that the solution is guaranteed to converge when we use a feedback path. This is because in the worst case we end up creating a new bus (to migrate components away from the violated bus) which
increases the cost of the system, but as a tradeoff we get improved system performance (even after we consider bridge overhead delays) and the ability to meet bus cycle time constraints.

Finally, after any violations have been resolved and all TCP constraints satisfied, we output the final synthesized bus topology, parameter values for bus speeds, data bus widths, DMA burst size and arbitration priority ordering, along with the feasible floorplan.

5.5 Case Studies

We applied our automated bus-based communication architecture synthesis approach on three industrial strength designs from the network communication domain. In the first case study, we selected a network communication SoC subsystem used for fast data packet processing and forwarding. Figure 5.9 shows the CTG for this system. There are two data manipulation related TCP constraints that must be satisfied in this system. The first TCP involves the encryption engine and includes the ARM926, ASIC1, RAM3 and EXT_IF blocks. The EXT_IF block fetches data and stores it in RAM3. The ASIC1 and ARM926 blocks fetch non overlapping sections of the data, process them and store them back in RAM3, from where the EXT_IF block fetches and streams them out at a minimum rate of 200 Mbps. The second TCP involves the USB subsystem. Data packets received at the USB are routed to RAM1. The ARM926 reads this data, processes it and stores it back to RAM1 from where the DMA engine transfers it to SDRAM_IF, which streams it out at a minimum rate of 480 Mbps. There is also a third subsystem which involves the SWITCH, RAM2 and ARM926 components. However, this is a very low
priority data path which has no data rate constraint from the designer, and therefore we
do not classify it as another TCP to be satisfied.

![Network Communication SoC Subsystem](image)

Figure 5.9 Network Communication SoC Subsystem

**Table 5.2 Customizable Parameter Set**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus width</td>
<td>8, 16, 32</td>
</tr>
<tr>
<td>Bus speed</td>
<td>33, 66, 100, 133, 166, 200</td>
</tr>
<tr>
<td>DMA burst size</td>
<td>1, 2, 4, 8, 16</td>
</tr>
<tr>
<td>Arbitration strategy</td>
<td>static priority</td>
</tr>
</tbody>
</table>

Table 5.2 shows the Communication Parameter Constraint set (Ψ) for this case study.
The target communication architecture for the automated synthesis is the AMBA2 high
performance AHB bus and a low bandwidth APB bus. For the floorplanner, we give
maximum priority to minimizing wire length for components on a bus, and equal lower priorities for area and total wire length minimization.

Figure 5.10 Synthesized SoC subsystem

Figure 5.11 Floorplan for SoC subsystem
Table 5.3 Communication Parameter Values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AHB1</td>
</tr>
<tr>
<td>Bus width</td>
<td>32</td>
</tr>
<tr>
<td>Bus speed</td>
<td>133</td>
</tr>
<tr>
<td>DMA burst size</td>
<td>16</td>
</tr>
<tr>
<td>Arbitration strategy</td>
<td>ARM&gt;USB&gt;DMA&gt;EXT_IF&gt;ASIC1&gt;SWITCH</td>
</tr>
</tbody>
</table>

Figure 5.10 shows the final output of our synthesis flow – a synthesized architecture which meets all throughput and timing constraints. The values for the generated communication parameters are given in Table 5.3 and the final floorplan for this system is shown in Figure 5.11. The automated synthesis engine initially created 2 AHB buses, with the SWITCH and RAM2 components connected to AHB1, which was assigned a clock speed of 200 Mhz to meet the encryption path throughput constraint. However, the floorplanning engine detected a cycle time violation for the bus due to excessive capacitive loading. The *topology_mutate* stage then split the shared AHB bus and assigned the ARM926, ASIC1 and EXT_IF masters and their associated slaves to one bus, and the SWITCH and RAM2 components to another AHB bus, to reduce capacitive loading. Finally, the *optimize_design* function reduced the bus speeds for the AHB buses from 200 Mhz to 133 Mhz and the APB bus to 66 Mhz, to lower the cost of the system. Both the throughput constraints were still met at these lower bus speeds. The synthesis engine made a simple assumption and assumed a 133 Mhz bus speed for AHB3 to simplify the design of BRIDGE3 to AHB1, but a designer can choose to further lower the AHB3 bus speed if a more complex bridge is acceptable.
For our second case study, we considered a derivative of the network communication subsystem from Figure 5.9, which extends and partially modifies the functionality of the previous system. Figure 5.12 shows this derivative architecture, which has an additional TCP constraint involving the ARM926, SWITCH, RAM2 and two newly added components: a memory array (RAM4) and an ASIC block (ASIC2). In this TCP, data packets received from the SWITCH are stored in RAM2. These packets are retrieved by
ASIC2 which reads and modifies some protocol header information before storing it back to RAM4 from where the SWITCH must stream it out at a minimum data rate of 3.2 Gbps. The ARM926 is used minimally, for directing data flow in this TCP.

Figure 5.13 Synthesized SoC subsystem for derivative architecture

Figure 5.14 Final floorplan for derivative SoC subsystem
Table 5.5 Communication Parameter Values (Derivative Arch.)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus width</td>
<td>32, 32, 64, 32</td>
</tr>
<tr>
<td>Bus speed</td>
<td>100, 100, 200, 66</td>
</tr>
<tr>
<td>DMA burst size</td>
<td>16</td>
</tr>
<tr>
<td>Arbitration strategy</td>
<td>SWITCH&gt;ASIC2&gt;ARM&gt;USB&gt;EXT_IF&gt;DMA&gt;ASIC1</td>
</tr>
</tbody>
</table>

The Communication Parameter Constraint set ($\Psi$) for this case study is shown in Table 5.4 and is slightly modified from Table 5.2, with the addition of a larger data bus width value of 64, to handle the increased bandwidth requirements. Also, instead of using the AMBA2 AHB bus architecture like in the previous case, we modify the target communication architecture to AMBA3 AXI. Our synthesis flow outputs the architecture shown in Figure 5.13. The values for the generated communication parameters are shown in Table 5.5 and the final floorplan is shown in Figure 5.14. Since AXI supports separate channels for reads and writes, the bus speeds required to maintain throughput are lower (100 MHz). The AXI3 bus which supports the SWITCH TCP has a 64 bit data width and a high 200 MHz bus clock speed in order to maintain the high data flow rate.

For the third case study, we chose a multi-processor system (MPSoC) networking subsystem. Figure 5.15 shows the CTG for the system. For clarity, the TCPs are presented separately in Table 5.6. The Communication Parameter Constraint set ($\Psi$) is shown in Table 5.7. The target communication architecture for the synthesis process is the AMBA2 AHB bus architecture.
Figure 5.15 MPSoC networking subsystem

Table 5.6 Throughput Constraint Paths (TCPs)

<table>
<thead>
<tr>
<th>IP cores in Throughput Constraint Path (TCP)</th>
<th>Throughput Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC1, Acc1, Network I/F2, MEM5, MEM6</td>
<td>1.2 Gbps</td>
</tr>
<tr>
<td>ARM2, MEM7, DMA, Network I/F1</td>
<td>360 Mbps</td>
</tr>
<tr>
<td>ASIC2, DMA, Network I/F3, MEM5, MEM1, SlvAcc</td>
<td>200 Mbps</td>
</tr>
<tr>
<td>ARM1, MEM2, DMA, SDRAM1, Network I/F1</td>
<td>640 Mbps</td>
</tr>
<tr>
<td>ASIC3, Network I/F1, MEM3, MEM4, Network I/F3</td>
<td>1 Gbps</td>
</tr>
</tbody>
</table>
Table 5.7 Customizable Parameter Set

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus width</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>Bus speed</td>
<td>50, 100, 150, 200, 250, 300</td>
</tr>
<tr>
<td>DMA burst size</td>
<td>1, 2, 4, 8, 16</td>
</tr>
<tr>
<td>Arbitration strategy</td>
<td>static priority</td>
</tr>
</tbody>
</table>

ARM1 is a protocol processor (PP) while ARM2 is a network processor (NP). The ARM1 PP is responsible for setting up and closing network connections, converting data from one protocol type to another and exchanging data with the NP using shared memory. The ARM2 NP directly interacts with the network ports and is used for assembling incoming packets into frames for the network connections, network port packet/cell flow control, keeping track of errors and gathering statistics. The ASIC1 block performs hardware cryptography acceleration while ASIC2 and ASIC3 are used for other data packet and frame processing. The DMA is used to handle fast memory to memory and network interface data transfers, freeing up the processors for more useful work.

The synthesis process first generated the system shown in Figure 5.16 (a). However, once we passed the architecture through the floorplanning and wire delay estimation stage, it was discovered that the system was not feasible because of the excessive cumulative load capacitance on the AHB1 bus, which caused a timing violation. Figure 5.16 (b) shows the floorplan layout for this configuration. The synthesis process records this violation, and re-synthesizes the communication architecture to come up with the
architecture shown in Figure 5.16 (c) with a reduced capacitive loading on AHB1 while still satisfying all TCP constraints. This architecture does not violate any bus cycle time constraints and the final floorplan is shown in Figure 5.16 (d). Note that the synthesis process splits the SDRAM2 and MEM4 components, moving portions of both these components to a local bus of the ARM2 processor. This reduces unnecessary traffic and capacitive loading on the shared AHB bus. The synthesized communication parameter values are shown in Table 5.8. Since most of the streamed data was native 32 bits, a higher 64 bit bus width did not affect the performance significantly and the synthesized buses all have 32 bit data bus widths.

![Diagram of intermediate configuration for MPSoC networking subsystem](image)

**Figure 5.16** Intermediate configuration for MPSoC networking subsystem
Figure 5.17 Floorplan for intermediate configuration

Figure 5.18 Final synthesized configuration for MPSoC networking subsystem
We now compare the quality of our synthesis process. Since none of the existing synthesis approaches is aimed at detecting bus cycle time violations early in the design flow, there is no direct point of comparison. We chose to compare the quality of our synthesized designs with an approach which maps all the components in the application to a single main/peripheral shared bus (initial), an automated bus architecture synthesis
flow which does not use a high level floorplanner (ABS) and a manually intensive, high level synthesis effort by a designer which also makes use of a floorplanning and wire delay estimation engine to detect timing violations (manual) just like our floorplan-aware automated bus architecture synthesis approach (FABSYN). The manual synthesis approach involves a designer manually selecting a combination of bus topology and communication parameter values, simulating the high level design models in SystemC and then iteratively modifying the bus architecture and parameter values based on the simulation results and designer intuition, till all constraints are found to be satisfied.

Table 5.9 Synthesis Result Comparison

<table>
<thead>
<tr>
<th>Case Study 1 Designs</th>
<th>initial</th>
<th>ABS</th>
<th>manual</th>
<th>FABSYN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Busses</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>TCP constraints satisfied</td>
<td>0/2</td>
<td>2/2, not feasible</td>
<td>2/2</td>
<td>2/2</td>
</tr>
<tr>
<td>Execution cycles (millions)</td>
<td>49.76</td>
<td>24.51</td>
<td>18.80</td>
<td>20.32</td>
</tr>
<tr>
<td>Time to synthesize</td>
<td>~mins</td>
<td>~hours</td>
<td>~days</td>
<td>~days</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case Study 2 Designs</th>
<th>initial</th>
<th>ABS</th>
<th>manual</th>
<th>FABSYN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Busses</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>TCP constraints satisfied</td>
<td>0/3</td>
<td>3/3, not feasible</td>
<td>3/3</td>
<td>3/3</td>
</tr>
<tr>
<td>Execution cycles (millions)</td>
<td>88.48</td>
<td>47.63</td>
<td>26.58</td>
<td>29.10</td>
</tr>
<tr>
<td>Time to synthesize</td>
<td>~mins</td>
<td>~hours</td>
<td>~days</td>
<td>~days</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case Study 3 Designs</th>
<th>initial</th>
<th>ABS</th>
<th>manual</th>
<th>FABSYN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Busses</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>TCP constraints satisfied</td>
<td>0/5</td>
<td>5/5, not feasible</td>
<td>5/5</td>
<td>5/5</td>
</tr>
<tr>
<td>Execution cycles (millions)</td>
<td>170.11</td>
<td>98.20</td>
<td>94.50</td>
<td>92.69</td>
</tr>
<tr>
<td>Time to synthesize</td>
<td>~mins</td>
<td>~hours</td>
<td>~days</td>
<td>~hours</td>
</tr>
</tbody>
</table>
Table 5.9 compares the results from our synthesis approach for the three case studies with the results from the other approaches. The *initial* approach is unable to satisfy any of the TCP constraints for all three of the case studies, because of excessive data traffic conflicts on its restricted number of buses. In contrast, the *ABS* approach does manage to satisfy TCP constraints for all the case studies, but in each case it synthesizes a bus architecture with bus clock cycle time violations that remain undetected, and thus the synthesized architecture is not feasible in each case. The *manual* approach satisfies all TCP constraints and is also able to detect and eliminate bus clock cycle time violations in the design, just like our *FABSYN* approach. However, there are a few key differences between the *manual* approach and our *FABSYN* approach. Firstly, the *manual* approach generates bus architectures having a greater implementation cost (i.e. having a larger number of buses) when compared with architectures generated using our approach. This is because our automated flow is able to traverse a much larger communication parameter exploration space than the *manual* approach, and prevents us from making conservative decisions to create a new bus like in the *manual* approach, unless all suitable combinations of communication parameters are unable to meet the TCP constraint for the existing bus topology. Secondly, the performance of the architecture generated by the *manual* approach is actually found to be better than our *FABSYN* approach (except for the third case study, where frequent bridge delay overheads reduce performance). This is because of the larger number of buses used by the *manual* approach, which reduces data traffic conflict and improves concurrency, at the cost of increasing the implementation cost. But it is important to note is that we’re not really concerned about the absolute performance of the system. What is important to us is that we satisfy all TCP constraints.
and minimize the implementation cost of the synthesized architecture, and that we do so in a reasonable amount of time. The manual approach suffers from the major drawback that it takes several days for the designer to come up with a bus architecture which is typically overdesigned and exceeds the requirements (resulting in a more expensive system), whereas our FABSYN approach generates a better quality architecture in a matter of a few hours.

5.6 Conclusion

In this chapter, we presented a technique for the physically-aware synthesis of on-chip communication architectures, and used it to introduce physical implementation awareness into our automated hierarchical shared bus-based on-chip communication architectures. We described the high level floorplanning and delay estimation engines to generate a layout of the components on the chip and detect bus cycle time violations early in the design flow, at the system level. Our synthesis framework synthesizes a low-cost bus topology and generates values for bus architecture parameters such as arbitration priority ordering, bus widths, bus speeds and a DMA burst size, required to meet the performance constraints in the design. Results from the automated synthesis of AMBA based bus architectures for the network communication subsystem case studies show the usefulness of our approach. Our physically-aware synthesis approach reduces the exploration and design time by at least an order of magnitude when compared to a manual effort, while also ensuring the feasibility of the synthesized physical design. Furthermore, our approach is easily portable across different standard on-chip communication architectures such as IBM CoreConnect and Sonics SMART Interconnect, and can be extended to
automatically synthesize other bus architecture specific parameters such as out-of-order (OO) buffer sizes as well.
Chapter 6
COSMECA: On-Chip Memory and Communication Architecture Co-synthesis

6.1 Introduction

In addition to the communication architecture, the memory architecture in MPSoCs has a significant impact on overall performance, especially in memory dominated applications. The memory blocks in an MPSoC can occupy up to 70% of the die area [93]. Estimates indicate that this figure will go up to 90% in the coming years [92]. Since memory and communication architectures have such a significant impact on system cost, performance and time-to-market, it becomes imperative for designers to focus on their exploration and synthesis early in the design flow, with the help of efficient design flow concepts such as those proposed in platform-based design [94].

Traditionally, in platform-based design, memory synthesis is performed before the communication architecture synthesis step [95] [96] [97] [98] [99]. While treating these two steps separately is done mainly due to tractability issues [100], it can lead to sub-optimal design decisions. Consider the example of a networking MPSoC subsystem shown in Figure 6.1 (a). The figure shows the system after HW/SW partitioning, with all the IPs defined, including memory which is synthesized based on data size and high-level bandwidth constraint analysis. Figure 6.1 (b) shows the traditional approach where communication architecture synthesis is performed after memory synthesis, while Figure
6.1 (c) shows the case where memory and communication architectures have been co-synthesized using the COSMECA co-synthesis approach [115] [116].

Figure 6.1 Comparison of memory and communication architecture synthesis approaches for (a) MPSoC example, (b) result of performing memory synthesis before communication architecture synthesis, and (c) result of performing co-synthesis of memory and communication architectures.
Now let us consider the implications of using a co-synthesis framework. Firstly, the co-synthesis approach is able to detect that the data arrays stored in Mem1 and Mem2 end up sharing the same bus, and automatically merges and then maps the arrays onto a larger single physical memory from the library, thus saving area. Secondly, the co-synthesis approach is able to merge data arrays stored in Mem3 and Mem5 onto a single memory from the library, saving not only area but also eliminating two buses, as shown in Figure 6.1 (c). However, Mem5 cannot share the same bus as Mem3 (or Mem4) in Figure 6.1 (b) because the access times of the pre-synthesized physical memories are such that they cause traffic conflicts which violate bandwidth constraints. Thirdly, due to the knowledge of support for out-of-order (OO) transaction completion by the communication architecture, the co-synthesis approach is able to add an OO buffer of depth 6 to Mem4, which enables it to reduce the number of ports from 2 to 1, thus saving area, while still meeting bandwidth constraints. It is thus apparent that the COSMECA co-synthesis approach is able to make better synthesis decisions by exploiting the synergy and interdependence between the memory and communication architecture design spaces, to reduce the overall cost of the synthesized system.

6.1.1 Chapter Overview

In this chapter, we present an automated application-specific co-synthesis framework for memory and communication architectures (COSMECA) for MPSoC designs. The framework extends our previous work on automated communication architecture synthesis (described in Chapter 4) by adding another dimension to it, enabling the synthesis of the memory subsystem concurrently with communication architecture
synthesis. In its essence, *COSMECA* is a novel memory and communication architecture co-synthesis methodology which improves upon existing synthesis approaches by (i) automatically generating bus topology and parameter values for arbitration schemes, bus speeds and OO buffer sizes, while considering dynamic simulation effects, and (ii) simultaneously determining a mapping of data arrays to physical memories while also deciding the number, size, ports and type of these memories, from a memory library. The framework tailors the memory and communication architectures to the application being considered, to reduce system cost. The primary objective is to design a communication architecture having the least number of buses, which satisfies performance and memory area constraints, while the secondary objective is to reduce the memory area cost. We consider a bus matrix type of on-chip communication architecture for synthesis, since it is increasingly being used by designers in high bandwidth designs today.

In Section 6.2, we present an overview of the on-chip memory subsystem that we consider in our co-synthesis framework. In Section 6.3, we describe our assumptions and the problem formulation for co-synthesis. In Section 6.4, we define the memory-communication parameter constraint set. In Section 6.5, we present details of our COSMECA co-synthesis flow. Finally, in Section 6.6, we present industrial strength MPSoC case studies on which we applied the COSMECA framework for on-chip memory and communication architecture co-synthesis.

### 6.1.2 Related Work

Previous research in the area of memory and communication architecture synthesis has either ignored the co-synthesis aspect, or focused on a small subset of the problem.
Typically, high-level synthesis approaches perform memory allocation and mapping before communication architecture synthesis [95] [96] [97] [98] [99], ignoring the overhead of the communication protocol during synthesis. While treating these two steps separately is mainly due to tractability issues [100], the merits of integrating communication synthesis with memory synthesis are clearly demonstrated by Shalan et al. [101]. Only a few approaches have attempted to simultaneously explore memory and communication subsystems. Shalan et al. [101] presents a tool to automatically generate a full crossbar and a dynamic memory management unit (DMMU). Grun et al. [102] consider the connectivity topology early in the design flow in conjunction with memory exploration, for simple processor-memory systems. Kim et al. [103] propose an approach that deals with bus topology and static priority based arbitration exploration, to determine the best memory port-to-bus mapping for pre-synthesized memory blocks. More recently, Srinivasan et al. [104] present an approach to simultaneously consider bus topology splitting and memory bank partitioning during synthesis. While they consider a limited design space compared to our approach (they do not consider the effect of communication parameters or different memory types), their focus is on the problem of system energy reduction, which is not currently addressed by our approach. Other approaches which deal with memory synthesis make use of static estimations of communication architectures such as those proposed by Knudsen et al. [105] and Nandi et al. [106]. Such approaches are unable to capture dynamic effects such as contention and address only a limited exploration space. More importantly, none of the abovementioned approaches attempts to perform co-synthesis like our framework.
6.2 Memory Subsystem

There are a variety of different memory types available to satisfy on-chip memory requirements in applications. Typically, designers have used off-chip DRAMs for larger memory requirements and on-chip embedded SRAMs for smaller memory requirements. Lately, on-chip embedded DRAMs are gaining in popularity as they eliminate I/O signals to separate memory chips, boosting performance and reducing noise, as well as pin count, which ends up lowering system cost. Although SRAMs have smaller access times than DRAMs, they also take up a larger area, requiring a tradeoff between area and performance between the two memory types during synthesis. The memory synthesis in COSMECA uses a memory library populated by on-chip SRAMs and on-chip DRAMs, having different capacities, areas, ports and access times. This information can be easily represented in the form of a table that captures different SRAM and DRAM variants and their characteristics. We assume that the word size of these memories is equal and fixed, based on the application. Data arrays and groups of scalars in the application are grouped together into virtual memories (VMs) based on certain rules, before being mapped onto the appropriate physical memories from the library, which allow the application to meet its area and performance constraints. The grouping of data blocks in our approach allows us to reduce the number of memories in the design, thus reducing area. We also try to avoid multi-port memories because of their excessive area and cost overhead.
6.3 Assumptions and Problem Formulation

We are given an application for which we assume the HW/SW partitioning has already been performed. The resulting MPSoC design has possibly several hardware and software IPs onto which application functionality has been mapped. Memory in this model is initially represented by abstract *data blocks* (DBs) which are collections of scalars or arrays accessed by the application, similar to *basic groups* in [98]. Generally, this MPSoC design will have performance constraints, dependent on the application. The *throughput* of communication between components is a good measure of the performance of a system. To represent performance constraints in COSMECA, we use a **Communication Throughput Graph** $CTG = G(V,A)$ which is a directed graph, where each vertex $v$ represents an IP (or DB) in the system, and an edge $a$ connects components that need to communicate with each other. A **Throughput Constraint Path** (TCP) is a sub-graph of a CTG, consisting of a single component for which data throughput must be maintained and other masters, slaves and DBs which are in the critical path that impacts the maintenance of the throughput. The problem definition is as follows:

**Problem Definition:** A bus $B$ can be considered to be a partition of the set of components $V$ in a CTG, where $B \subseteq V$. Then our primary objective is to determine an optimal component to bus assignment for a bus matrix architecture, such that the partitioning of $V$ onto $N$ buses results in a minimal number of buses $N$ and satisfies memory area bounds while meeting all performance constraints in the design, represented by the TCPs in a CTG. As a secondary objective, we attempt to reduce memory area cost of the solution.
6.4 Communication-Memory Parameter Constraint Set $\Psi$

In the interest of generating a practically realizable system, we allow a designer to specify a discrete set of valid values (referred to as a constraint set $\Psi$) for communication parameters such as bus clock speeds, OOC buffer sizes and arbitration schemes. Additionally, $\Psi$ allows the specification of constraints on the type of memory to allocate for DBs, for instance, in the case of a DB which the designer knows must be read from an EEPROM memory. We allow the specification of two types of constraint sets for components – a global constraint set ($\Psi_G$) and a local constraint set ($\Psi_L$). The presence of a local constraint overrides the global constraint, while the absence of it results in the resource inheriting global constraints. For instance, a designer might set the allowable bus clock speeds for a set of buses in a subsystem to multiples of 33 MHz, with a maximum speed of 166 MHz, based on the operation frequency of the cores in the subsystem, while globally, the allowed bus clock speeds are multiples of 50 MHz, up to maximum of 250 MHz. This provides a convenient mechanism for the designer to bias the co-synthesis process based on knowledge of the design and the technology being targeted. Such knowledge about the design is not a prerequisite for using our co-synthesis framework, but informed decisions can help avoid the synthesis of unrealistic system configurations. The size of the constraint set directly affects the time taken for co-synthesis. The larger the number of values for the communication parameter constraints in the set (e.g., larger number of allowed bus speeds), the longer it takes for the co-synthesis framework to arrive at a solution since a larger design space has to be considered. However, the larger the number of memory mapping constraints in the set,
the lesser is the amount of time taken to arrive at a solution, since the memory mappings to be considered during co-synthesis are now reduced.

6.5 COSMECA Co-Synthesis Flow

We describe the COSMECA co-synthesis flow in more detail in this section. Figure 6.2 gives a high level overview of the flow. The inputs to COSMECA include a Communication Throughput Graph (CTG), a library of behavioral IP models (IP library) and memory models (mem library – captured with a table that has different SRAM and DRAM variants, each with corresponding capacity, area, number of ports, and access time characteristics), a Data Block Dependency Graph (DBDG), a target bus matrix template (e.g., AMBA bus matrix) and a communication-memory constraint set (Ψ) – which includes ΨG and ΨL. The general idea is to first preprocess the memory (represented by DBs in the CTG) in the design by merging the non conflicting DBs into virtual memory (VM) blocks to reduce memory cost. Then we map the modified CTG to a full bus matrix template and optimize the matrix by removing unused buses. Next, we perform a static branch and bound hierarchical clustering of slave components in the matrix which further reduces the number of buses, and store prospective matrix architecture solutions in a ranked matrix solution database. We then use a heuristic (memmap), which first merges VMs at each slave access point (SAP) in the bus matrix to further reduce memory cost and then maps these VMs to physical memory modules from the memory library. The output of memmap is a set of N valid solutions which meet memory area and performance constraints. Finally we optimize the output solutions to
reduce bus speeds, arbitration costs and prune out-of-order (OO) buffer sizes. We now elaborate on the five phases in the COSMECA flow, shown in Figure 6.2.

**Phase 1. mem preprocess:** In the first phase, we merge data blocks (DBs) in the CTG into virtual memories (VMs) to reduce memory area cost, by potentially reducing the number of memory modules in the system. Only DBs satisfying the two criteria of having (i) similar edges (i.e. edges from the same masters) and (ii) non-overlapping access are merged, so as not to constrain the mapping freedom and eliminate useful channel clustering possibilities later in the flow. Figure 6.3 (a) shows a CTG for an example MPSoC system, with the following groups of DBs having similar edges: (DB1, DB2) and (DB4, DB5, DB6). We use a Data Block Dependency Graph (DBDG) to determine if DBs have non-overlapping access. The DBDG is a directed graph which shows the
dependency of DB accesses on each other. It can either be created manually or derived automatically from a Control Data Flow Graph (CDFG). A node in a DBDG represents a DB access while an edge represents a dependency between DBs – a DB cannot be accessed till the source DBs of all its input edges have been accessed. Figure 6.3 (b) shows the DBDG for the example in Figure 6.3 (a). If two DBs have similar edges and non-overlapping access, they are eligible for merger (e.g., DB1, DB2 in Figure 6.3 (b)). The size of the VM created, after merger, depends on the lifetime analysis of merged DBs – it is the sum of the sizes of the merged DBs, unless the lifetimes do not overlap, in which case it is the size of the larger DB being merged. Figure 6.3 (b) shows the lifetime of DB1. It is possible for DB2 to overwrite DB1, thus saving memory space.

**Phase 2. matrix map and analyze:** In the second phase, the modified CTG is mapped onto a full bus matrix template. The full bus matrix is subsequently pruned by removing unused buses on which there are no data transfers. Dedicated slave and memory components are also migrated to the local buses of their corresponding masters to further reduce buses in the matrix. Figure 6.3 (d) shows the bus matrix after these steps, for the example in Figure 6.3 (a). Finally, we perform a fast high level, Transaction Level (TLM) simulation of the application, using communication protocol-independent channels for communication and assuming no arbitration contention, to obtain application-specific data traffic statistics such as the number of transactions on a bus and average transaction burst size on a bus.
Figure 6.3 COSMECA co-synthesis example
Knowing the bandwidth to be maintained on a bus from the TCPs in the CTG, we can also estimate the minimum clock speed at which any bus in the matrix must operate, in order to meet its throughput constraint, as follows. The data throughput ($\Gamma_{TLM/B}$) from the TLM simulation, for any bus $B$ in the matrix is given by

$$\Gamma_{TLM/B} = (numT_B \times sizeT_B \times width_B \times \Omega_B) / \sigma$$

where $numT$ is the number of data transactions on bus $B$, $sizeT$ is the average data transaction size, $width$ is the bus width, $\Omega$ is the clock speed, and $\sigma$ is the total number of cycles of TLM simulation for the application. The values for $numT$, $sizeT$ and $\sigma$ are obtained from the TLM simulation. To meet throughput constraint $\Gamma_{TCP/B}$ for bus $B$,

$$\Gamma_{TLM/B} \geq \Gamma_{TCP/B}$$

$$\therefore \Omega_B \geq (\sigma \times \Gamma_{TCP/B}) / (numT_B \times sizeT_B \times width_B)$$

The minimum bus speed thus found is used to create (or update) the local bus speed constraint set $\Psi_{L(speed)}$ for bus $B$.

**Phase 3. Branch and bound clustering algorithm:** In the third phase, a *static branch and bound hierarchical clustering algorithm* is used to cluster slave/memory components to reduce the number of buses in the matrix even further. This step is similar to that described in our synthesis flow in Chapter 4, and will not be described here again. The output of this phase is a set of solutions that are subsequently ranked from best (least
number of buses) to worst and stored in a ranked matrix solution database. Figure 6.3 (e) shows the best solution after this phase, for the example in Figure 6.3 (a). For each of the solutions, we set OO buffer sizes to the maximum allowed in $\Psi$, for the components which support it. For the arbitration scheme at the slave access points (SAPs) of the bus matrix, we initially use a possibly more expensive-to-implement arbitration strategy such as the TDMA/RR scheme to proportionally grant accesses to masters based on the magnitude of throughput requirements.

**Phase 4. memmap heuristic:** In the next phase, we use the memmap heuristic to guide the mapping of VMs to physical memories in the memory library. Figure 6.4 shows the pseudo code for the memmap heuristic. The goal is to find $N$ solutions which satisfy memory area and performance constraints of the design. We begin by selecting the best solution from the ranked matrix solution database, populated in the previous phase, and simulate the design (Lines 3-4), with the CCATB simulation engine described earlier in Chapter 2. The output of this simulation is a set of memory access traces which are used to determine the extent of access overlap of VMs at each SAP. If the overlap is below a user defined overlap threshold $\tau$, we merge the VMs (Lines 5-6). Figure 6.3 (e) shows how we merge $VM2$ and $VM3$, as their memory access trace shown in Figure 6.3 (c) has an overlap less than the chosen value for $\tau$. The size of the merged VM is the sum of the memory sizes, unless the lifetimes do not overlap, in which case it is the size of the larger of the two VMs being merged. This VM merge step further reduces the number of memories, and consequently memory area cost.
1: **procedure** memmap()
2: **while** (num_sol < N) **do**
3:   select next candidate from ranked matrix solution database
4:   simulate design; //to generate memory trace
5:   **for each** SAP **do**
6:      merge VMs with overlap ≤ τ %
7:   **for each** VM **do**
8:      if (VM data overlap ≤ τ %)
9:         map to single port physical mem with best size match, max. port b/w
10:    else
11:       map to dual port physical memory with best size match, max. port b/w
12:      simulate design; //to verify mem area, performance constraint satisfaction
13:     if (performance constraint violation) **then**
14:        remove candidate from ranked matrix solution database; goto 3
15:     else **if** ((perf. constraint satisfied) && (mem area constraint satisfied)) **then**
16:        add to final solution database; num_sol++
17:     area_improvement_possible = true
18:   **while** ((num_sol < N) && (area_improvement_possible)) **do**
19:      **for each** SAP **do**
20:         randomly select eligible VM
21:         map physical memory with best size, port match, lower area
22:         simulate design; //to verify area, performance constraint satisfaction
23:      if ((perf constraint satisfied) && (mem area constraint satisfied)) **then**
24:         add to final solution database; num_sol++
25:    else
26:       undo mapping for VM with port bandwidth violation
27:       make VM with violation ineligible for further selection
28:     if (all VMs ineligible) **then**
29:        area_improvement_possible = false
30: **end** memmap

**Figure 6.4 memmap heuristic**

Next we proceed to map the VMs in the design to physical memories from the memory library (Lines 8-11). We choose the best memory from the library which fits the size requirement and has the maximum port bandwidth (i.e., combination of access time and operating frequency, which determines performance, expressed in terms of port bandwidth). The mapping step takes into consideration any memory mapping constraints in \( \Psi \). It is possible that a VM has self conflict greater than \( \tau \), in which case we map a dual
port memory if possible, otherwise we use single port memories. The type of port (R, W, R/W) is determined by the maximum simultaneous reads/writes from the memory trace. The reason for using physical memories with the best performance is that we want to check the feasibility of the matrix solution being considered, and eliminate a solution quickly if it is not a good match. Once the mapping is complete, we simulate the design. If throughput constraints are not met even for the memory mapping with best performance, we discard the matrix solution, and go back to select the next best matrix solution from the ranked matrix solution database. If performance constraints are met, we check if memory area constraints are met. If the area constraint is also met, we add the solution to the final solution database (Lines 12-16). Next, we attempt to lower memory area, while still meeting performance constraints, by changing the memory mapping for the current matrix solution (Lines 17-29). We do this by selecting one eligible VM at each SAP randomly and replacing the mapped physical memory with one which meets the size (capacity) requirements, but has lower area. All VMs are initially eligible for this mapping optimization. Next we simulate the design. If we find a performance violation at one or more SAPs, we undo the change in mapping for the VM at each violated SAP, and make it ineligible for further mapping optimization. The reason for selecting just one VM per SAP is that it makes it easier to determine which physical memory to VM mapping caused a performance violation, if one is found. If there is no performance violation, and if the area bounds are met, we have found a solution. We keep repeating this process till all VMs become ineligible for mapping optimization, or if the required N solutions have been found. If we encounter the former case and the number of
solutions found is less than \( N \), we proceed to select the next best solution from the ranked matrix solution database (Line 3), and repeat the process.

**Phase 5. optimize design:** Finally, we call the *optimize design* procedure for each of the \( N \) solutions obtained in the last phase. This simple procedure attempts to further reduce system cost by minimizing *(i)* bus speeds, *(ii)* arbitration scheme implementation cost and *(iii)* fix OO buffer sizes. The procedure first iterates over the buses in a solution, reducing the bus speed to the lowest possible allowed, simulating the design to ensure that no performance constraints are violated. Similarly, the procedure attempts to iteratively replace an arbitration scheme which is more expensive to implement (e.g., \( TDMA/RR \)) with one which is less expensive to implement (e.g., a static priority based scheme with priorities assigned depending on bandwidth requirements) at each SAP. Finally we fix the OO buffer sizes wherever applicable to the maximum number of buffers used during simulation of the application, if the number is less than the maximum allowed buffer size.

**6.6 Case Studies**

We applied the *COSMECA* approach to four industrial strength MPSoC applications – PYTHON, SIRIUS, VIPER2 and HNET8 – from the networking domain. PYTHON and SIRIUS are variants of existing industrial strength designs, VIPER2 and HNET8 are larger systems which have been derived from the next generation of MPSoC applications currently in development. Table 6.1 shows the number of components in each of these applications, after HW/SW partitioning. Note that the *Masters* column includes the
processors in the design, while the *Slaves* column does not include the memory blocks, which will be co-synthesized with the communication architecture later. While this simulation speed of our system-level modeling abstraction is fast for the amount of detail that it captures, modern MPSoC applications such as the ones we consider can still take several hours to simulate in their entirety. In order to reduce this overhead, we make use of representative testbenches for each of these applications, which capture the critical portions of the application functionality in a shorter execution time.

### Table 6.1 Core Distribution in MPSOC Applications

<table>
<thead>
<tr>
<th>Applications</th>
<th>Processors</th>
<th>Masters</th>
<th>Slaves</th>
</tr>
</thead>
<tbody>
<tr>
<td>PYTHON</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>SIRIUS</td>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>VIPER2</td>
<td>5</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>HNET8</td>
<td>8</td>
<td>13</td>
<td>17</td>
</tr>
</tbody>
</table>

We will first consider the PYTHON MPSoC and make use of the *COSMECA* co-synthesis framework to synthesize memory and communication architectures for it. Figure 6.5 shows the CTG for the PYTHON application, after the initial memory preprocessing phase in which DBs are merged into VMs. Not shown in the CTG, but included in our memory area analysis are the 32 KB instruction and data caches for each of the two processors. For clarity, the TCPs are presented separately in Table 6.2.
μP1 is used for overall system control, generating data cells for signaling, operating and maintenance, communicating and controlling external hardware and to setup and close data stream connections. μP2 interacts with data streams from external interfaces and performs data packet/frame encryption and compression. These processors interact with each other via shared memory and a set of shared registers (not shown here). The DMA engine is used to handle fast memory to memory and network interface data transfers, freeing up the processors for more useful work. PYTHON also has several

---

**Figure 6.5 PYTHON Communication Throughput Graph (CTG)**

**Table 6.2 PYTHON Throughput Constraint Paths (TCPs)**

<table>
<thead>
<tr>
<th>IP cores in Throughput Constraint Path (TCP)</th>
<th>TCP constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>μP2, VM2, VM3, Network I/F1, DMA, VM6</td>
<td>400 Mbps</td>
</tr>
<tr>
<td>μP2, VM2, VM6, VM7, DMA, Network I/F2</td>
<td>960 Mbps</td>
</tr>
<tr>
<td>μP1, MFSU, VM3, VM4, DMA, Network I/F1</td>
<td>400 Mbps</td>
</tr>
<tr>
<td>μP2, VM4, VM5, VM7, DMA, Network I/F1, Network I/F2</td>
<td>600 Mbps</td>
</tr>
</tbody>
</table>
peripherals such as a multi functional serial port interface (MFSU), a universal asynchronous receiver /transmitter block (UART), a general purpose I/O block (GPIO), timers (Timer, Watchdog), an interrupt controller (ITC) and two proprietary external network interfaces.

Table 6.3 PYTHON Global Constraint Set \( \Psi_G \)

<table>
<thead>
<tr>
<th>Set</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus speed</td>
<td>25, 50, 100, 200, 300, 400</td>
</tr>
<tr>
<td>arbitration strategy</td>
<td>static, RR, TDMA/RR</td>
</tr>
<tr>
<td>OO buffer size</td>
<td>1 – 8</td>
</tr>
<tr>
<td>mem mapping</td>
<td>VM1=&gt;EEPROM</td>
</tr>
</tbody>
</table>

Table 6.3 shows the global constraint set \( \Psi_G \) for PYTHON. For the synthesis we target an AMBA3 AX bus matrix. We assume a fixed bus width of 32 bits, as per application requirements. The memory area constraint is set to 120 mm\(^2\) and the estimated memory area numbers are for a 0.18-\(\mu\)m technology. We assume the value for overlap threshold \( \tau = 10\% \) for this example. Figure 6.6 shows the best solution (least number of buses) with the least memory area for PYTHON. The figure also shows bus speeds, memory sizes, number of ports and OO buffer sizes.
Figure 6.6 synthesizes output for PYTHON.

Figure 6.7 shows the variation in memory area and number of buses in the matrix for the ten best solutions (N=10), for PYTHON. From the figure we can see that no solution having 7 buses in the bus matrix exists for PYTHON. The dotted line indicates the solution shown in Figure 6.6. We can see that there is a significant variation of
combinations of memory area and number of buses, in the solution space. COSMECA thus allows a designer to tradeoff memory area and bus count during the solution selection process.

**Figure 6.8 Effect of varying threshold value on solution quality for PYTHON**

During the course of the COSMECA co-synthesis flow, we made use of a threshold factor \( \tau \) (Figure 6.4; memmap heuristic) to determine the extent to which virtual memories are merged at SAPs in the bus matrix. This parameter is specified by the designer. To understand the effect of this threshold factor \( \tau \) on the quality of solution, we varied the threshold value and repeated our COSMECA co-synthesis flow for the PYTHON MPSoC. The result of this experiment is shown in Figure 6.8. It can be seen that for very low values of \( \tau \) (e.g., < 10%), the number of buses in the matrix for the best solution is high. This is because low values of \( \tau \) discourage merger of virtual memories, which ends up creating a system with several physical memories that exceed memory area bounds due to their excessive area overhead. For larger values of \( \tau \) (e.g., \( \geq 20\% \)), the number of buses for the best solution is also high, because it becomes harder to meet
application throughput constraints with the large overlap. There might be slight variations
to this trend, depending upon a complex amalgamation of factors such as stringency of
throughput requirements, allowed maximum bus speeds, available memory port
bandwidths and data traffic schedules for the application. Typically however, for the
COSMECA co-synthesis framework, our experience shows that lower values around 10 –
20% for overlap threshold $\tau$ give the best quality solutions.

Figure 6.9 SIRIUS Communication Throughput Graph (CTG)
Next we consider a more complex application: the SIRIUS MPSoC, and go into more detail of how it was used as another driver for the \textit{COSMECA} framework. Figure 6.9 shows the CTG for the SIRIUS application, after the initial memory preprocessing phase in which DBs are merged into VMs. Not shown in the CTG, but included in our memory area analysis are the 32 KB instruction and data caches for each of the three processors. For clarity, the TCPs are presented separately in Table 6.4. \(\mu P1\) is a protocol processor (PP) while \(\mu P2\) and \(\mu P3\) are network processors (NP). The \(\mu P1\) PP is responsible for setting up and closing network connections, converting data from one protocol type to another, generating data frames for signaling, operating and maintenance and exchanging data with NP using shared memory. The \(\mu P2\) and \(\mu P3\) NPs directly interact with the network ports and are used for assembling incoming packets into frames for the network connections, network port packet/cell flow control, assembling incoming packets/cells into frames, segmenting outgoing frames into packets/cells, keeping track of errors and gathering statistics. ASIC1 performs hardware cryptography acceleration for DES, 3DES and AES. The DMA is used to handle fast memory to memory and network interface data transfers, freeing up the processors for more useful work. SIRIUS also has a number of

\begin{table}[h]
\centering
\begin{tabular}{|l|c|}
\hline
\textbf{IP cores in Throughput Constraint Path (TCP)} & \textbf{TCP constraint} \\
\hline
\(\mu P1, VM3, VM4, DMA, VM16, VM17, VM18\) & 640 Mbps \\
\(\mu P1, VM5, VM6, VM14, VM15, DMA, Network I/F2\) & 480 Mbps \\
\(\mu P2, Network I/F1, VM8, VM9\) & 5.2 Gbps \\
\(\mu P2, VM10, VM11, VM12, DMA, Network I/F3\) & 1.4 Gbps \\
\(ASIC1, \mu P3, VM16, VM17, VM18, Acc1, VM13, Network I/F2\) & 240 Mbps \\
\(\mu P3, DMA, Network I/F3, VM13\) & 2.8 Gbps \\
\hline
\end{tabular}
\caption{SIRIUS Throughput Constraint Paths (TCPs)}
\end{table}
network interfaces and peripherals such as interrupt controllers (ITC1, ITC2), a UART, timers (Watchdog, Timer1, Timer2) and a packet accelerator (Acc1).

**Table 6.5 SIRIUS Global Constraint Set \( \Psi_G \)**

<table>
<thead>
<tr>
<th>Set</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus speed</td>
<td>25, 50, 100, 200, 300, 400</td>
</tr>
<tr>
<td>arbitration strategy</td>
<td>static, RR, TDMA/RR</td>
</tr>
<tr>
<td>OO buffer size</td>
<td>1 – 8</td>
</tr>
<tr>
<td>mem mapping</td>
<td>VM16, VM17=&gt;DRAM; VM1, VM2=&gt;EEPROM</td>
</tr>
</tbody>
</table>

**Figure 6.10 Synthesized output for SIRIUS**

Table 6.5 shows the global constraint set \( \Psi_G \) for SIRIUS. For the synthesis we target an AMBA3 AXI bus matrix. We assume a fixed bus width of 32 bits, as per application requirements. The memory area constraint is set to 225 mm\(^2\) and the estimated memory
area numbers are for a 0.18-µm technology. We assume the value for overlap threshold $\tau = 10\%$ for this example. Figure 6.10 shows the best solution (least number of buses) with the least memory area for SIRIUS. The figure also shows bus speeds, memory sizes, number of ports and OO buffer sizes.

It should be noted that while COSMECA allows a designer the flexibility to assign variable bus clock frequencies for the buses in the matrix, this entails an overhead in the form of frequency converters at the interfaces (which might use buffering for timing isolation). As an alternative, a single low frequency for all the buses in the matrix is usually practically insufficient to meet high throughput requirements. In contrast, a higher fixed frequency for the entire matrix can end up dissipating excessive power in the bus logic and bus wires. So a designer needs to be aware of this tradeoff. Note that COSMECA can be made to synthesize a matrix with either different or one fixed bus clock frequency for the buses in the matrix.

![Figure 6.11 SIRIUS final solution space (for N=10)](image)

Figure 6.11 SIRIUS final solution space (for N=10)
Figure 6.11 shows the variation in memory area and number of buses for the ten best solutions \((N=10)\) for SIRIUS. The dotted line indicates the solution shown in Figure 6.10. It can be seen that the memory area cost varies dramatically, not only when the bus matrix configuration is changed (by changing number of buses), but also for the same configuration, for different memory mapping decisions. Again, the key observation from this experiment is that COSMECA enables a designer to select a solution having the desired tradeoff between memory area and bus count in the matrix.

![Figure 6.12 Effect of varying threshold value on solution quality for SIRIUS](image)

**Figure 6.12 Effect of varying threshold value on solution quality for SIRIUS**

To determine the impact of varying the threshold factor \(\tau\) on the quality of solution for the SIRIUS MPSoC, we varied the threshold value and repeated our COSMECA co-synthesis flow for SIRIUS. The result of this experiment is shown in Figure 6.12. The trend for this experiment is similar to our observation for Figure 6.8, which showed the results for this experiment on the PYTHON MPSoC. As observed earlier, lower values
around 10 – 20% for overlap threshold $\tau$ give the best quality solutions for the SIRIUS application.

**Table 6.6 Co-Synthesis Time for MPSOC Applications**

<table>
<thead>
<tr>
<th>Applications</th>
<th>Simulation runs</th>
<th>Total co-synthesis time (in hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PYTHON</td>
<td>13</td>
<td>3.5</td>
</tr>
<tr>
<td>SIRIUS</td>
<td>19</td>
<td>8.6</td>
</tr>
<tr>
<td>VIPER2</td>
<td>26</td>
<td>17.8</td>
</tr>
<tr>
<td>HNET8</td>
<td>38</td>
<td>28.5</td>
</tr>
</tbody>
</table>

We now present results for the number of simulation runs and total time taken during co-synthesis. Table 6.6 shows the total number of simulation runs and total simulation time in hours for the MPSoC applications. Note that the contribution of the static estimation phases such as the branch and bound clustering is almost negligible and simulation takes up most of the time during co-synthesis. It can be seen that the entire COSMECA flow took in the order of hours to generate the best solution for each of the four MPSoC applications considered. This is in contrast to the traditional semi-automated (or manual) communication architecture synthesis techniques which can take several days, and would take even longer with the added complexity of handling memory synthesis.

Finally, Figures 6.13 and 6.14 compare the number of buses and memory areas for the best solution (having least number of buses, minimum memory area for the solution) obtained with COSMECA and the traditional approach (where memory synthesis is done before communication architecture synthesis) for the four applications. It can be seen that COSMECA performs much better for each of the applications, saving from 25-40% in the number of buses in the matrix and from 17-29% in memory area, because it is able to
make better decisions by taking the communication architecture into account while allocating and mapping data blocks to physical memory components.

Figure 6.13 Comparison of best solution bus count

Figure 6.14 Comparison of best solution memory area
6.7 Conclusion

In this chapter, we have presented an automated application specific framework to co-synthesize memory and communication architectures (COSMECA) for MPSoC designs. COSMECA couples the decision making process during memory and communication architecture synthesis, which enables it to generate a lower cost system. The primary objective of the framework described in this chapter is to design a communication architecture having the least number of buses, which satisfies performance and memory area constraints, while the secondary objective is to reduce the memory area cost. Results of applying COSMECA to several industrial strength MPSoC applications from the networking domain indicate a saving of as much as 40% in number of buses and 29% in memory area compared to the traditional approach, where memory synthesis is performed before communication architecture synthesis. Note that the memory library model used consists of a table of different on-chip SRAM and DRAM variants that have different capacity, area, ports, and latency characteristics. While somewhat simplistic, this model still allows us to demonstrate the capabilities of the COSMECA framework. Designers can extend this library model in COSMECA to account for memories with more complex protocols and access time dynamics.
Chapter 7
Conclusions and Future Work

There is no argument that the exploration and design of on-chip communication architectures is one of the most important problems in an MPSoC design flow. On-chip communication architectures are the source of significant performance bottlenecks, dissipate a major chunk of the overall system power, have increasingly large area footprints, and critically impact time-to-market of MPSoC designs because of the enormous amount of time required for their exploration and implementation. As a result, it becomes essential to explore on-chip communication architectures as early as possible in the design flow, using tools and techniques for traversing the vast communication architecture design space. This thesis presented the novel COMMSYN framework for the automated and comprehensive synthesis of on-chip communication architectures at the system level. This chapter draws the conclusions from the research results obtained, and looks at some future research directions in the area of on-chip communication architecture design and exploration.

7.1 Conclusions

This thesis proposes the COMMSYN synthesis framework that is comprised of tools, methodologies and techniques to address several issues related to the modeling, exploration and synthesis of on-chip communication architectures for emerging MPSoC
applications. The major research contributions of this thesis can be classified in the context of the following five key problem areas:

- **Modeling for Fast Simulation and Rapid Prototyping of Communication Architectures**: The exploration of on-chip communication architectures as well as the entire MPSoC design early in the design flow requires the creation of a fast and accurate simulation model that can efficiently capture the behavior of hardware and software components in an MPSoC design. To address this challenge, we developed a novel modeling abstraction called CCATB that not only allows rapid system prototyping, but also enables fast and accurate on-chip communication architecture exploration. The CCATB kernel effectively aggregates delays during simulation, reducing event scheduling overhead and speeding up execution. We showed how CCATB models can be effectively incorporated into a traditional MPSoC design flow, and drastically reduce modeling, simulation and exploration time for industrial MPSoC designs.

- **Fast and Accurate Early Power Estimation for Communication Architectures**: On-chip communication architectures have a considerable impact on MPSoC power consumption (anywhere between 20-50% of overall system power). Thus there is a need to create models for estimating power consumption of on-chip communication architectures as early as possible in a design flow, for better planning and design decisions. Several existing approaches have proposed using detailed RTL or gate-level power models that are highly accurate, but are
also too time consuming to create and obtain power numbers with. Other high level approaches for communication architecture power estimation have fast estimation speeds but lower accuracy. Additionally, none of the existing approaches enable fast power model generation across multiple technology libraries. We developed a highly accurate power estimation approach, based on energy macro modeling, which enables comprehensive communication architecture power estimation, taking into consideration contributions from wires and communication architecture logic components such as arbiters, decoders, and buffers. These models can be plugged into system level simulation models for highly accurate (within 5% of gate-level estimates) and fast (as much as 2000× speedup over gate-level estimation) communication architecture power estimation. Additionally, it takes very little time to retarget the developed models to a new technology library, while maintaining estimation accuracy.

➢ Automated Communication Architecture Synthesis with Multiple Design Constraints: The on-chip communication architecture design space for emerging MPSoC applications has become too large to traverse efficiently with traditionally used manual or partially automated techniques. Existing synthesis techniques are limited to generating either the topology or protocol parameters for a fixed topology, but not both. They also do not optimize for a combination of multiple design goals such as power, performance and cost. To address these challenges, we proposed a comprehensive, fully automated synthesis framework that generates a communication architecture customized for an MPSoC application.
We developed heuristics to effectively prune the massive design space, and made use of optimization algorithms and estimation techniques to synthesize MPSoC communication architectures. Unlike existing approaches, the framework not only synthesizes the topology structure but also generates values for protocol parameters such as bus clock frequencies, arbitration schemes and buffer sizes, while optimizing for varying design goals such as power, performance and cost.

➢ **Physical Implementation Aware Early Communication Architecture Design:**

Techniques to synthesize on-chip communication architectures early in the design flow typically do not consider physical implementation issues that can arise late in the design flow, during the floorplanning, and place and route phases. As a result, there is a vast disconnect between a communication architecture synthesized by existing approaches early in the design flow, and the feasibility of its physical implementation. In order to address this challenge, we proposed a framework that introduces physical awareness into the communication architecture synthesis process, early in the design flow. The framework makes use of a simulated annealing floorplanner to create an early MPSoC floorplan, and uses estimation techniques to automatically detect and eliminate clock cycle timing violations during synthesis. Experimental results on several MPSoC designs showed the superiority of such a physically-aware synthesis framework in reducing design cycle time, compared to existing approaches that rely on physical design much later in the design process.
Memory-Communication Architecture Co-synthesis: In addition to the communication architecture, the memory subsystem is an important component of MPSoC designs. Estimates indicate that the chip area occupied by the memory architecture will go up to 90% in the coming years. Traditionally, memory synthesis is performed before the communication architecture synthesis step. While treating these two steps separately is done mainly due to tractability issues, it can lead to sub-optimal design decisions. We developed a novel framework that automatically co-synthesizes the memory and communication architectures. We proposed heuristics to reduce the number of communication architecture interconnections, and efficiently select the memory components in order to reduce the overall area of the MPSoC design, while still meeting performance constraints. The results of applying the framework to MPSoC applications from the networking domain indicated a saving of as much as 25% in chip area, when compared to the traditional approach of separate synthesis.

The solutions to the key problems areas described above are all encompassed in the COMMSYN automated synthesis framework. COMMSYN incorporates our fast and accurate simulation and power estimation models for early performance and power exploration. These exploration results are used together with our methodologies for physically-aware and memory-communication architecture co-design, to create a comprehensive, fully automated on-chip communication architecture synthesis framework that improves upon the state-of-the-art. Results of applying our framework on several industrial strength case studies have shown how the multi-faceted COMMSYN
framework accrues many benefits for MPSoC designs: improved design reliability and quality, better complexity management, reduced system cost and a faster time-to-market.

### 7.2 Future Directions

The design of on-chip communication architectures is a challenging problem for emerging MPSoC designs. While this thesis investigated several problems in the areas of modeling, exploration and synthesis of on-chip communication architectures, there are many more problems arising due to advances in process technology and increasing levels of component integration in MPSoC designs. The work presented in this thesis can be extended in the following directions:

- With increasing application complexity and with the advent of networks-on-chip (NoC), faster and more efficient simulation techniques are needed to perform exploration of large scale on-chip communication architectures. Techniques that make use of partial FPGA emulation together with software simulations are one of the many promising directions for future research, to improve exploration speed.

- In Ultra DSM (UDSM) technologies, estimating power for on-chip communication architectures will become increasingly challenging because of process, voltage and temperature (PVT) variations. These variations are due to an increase in leakage power and the use of power-aware design methodologies such as voltage islands, dynamic voltage/frequency scaling, etc. As a result of these
variations, it is very hard to predict the power dissipation of an on-chip communication architecture fabric. Our preliminary work has shown that there is a large variations in power dissipation (as much as 10×) of on-chip communication architectures under PVT variations [107]. We have taken the first step to propose PVT variation-aware power estimation of on-chip communication architectures at the system level. A lot more work is needed to accurately account for all the sources of variation, including on-die variations, for accurate power estimation of on-chip communication architectures.

The automated synthesis of on-chip communication architectures will become more complex in the next few years, as large MPSoC designs consisting of hundreds of components become the norm. There are two directions in which our research on the synthesis of on-chip communication architectures can be extended. Firstly, research is needed to develop communication architecture fabrics that can handle the communication needs of emerging applications. NoCs have received a lot of interest of late and many researchers consider them to be a viable solution that can provide scalable bandwidth and performance required. But NoCs have drawbacks too – such as taking up a large area footprint, low latency response and high power consumption. It would be beneficial to explore communication architectures such as hybrid bus-NoC configurations and develop tools for the automated exploration and synthesis of these architectures. Secondly, future research in this area must consider several additional design constraints such as temperature, security and reliability when deciding on which communication
architecture configuration to select, since these constraints are becoming just as important as the traditional constraints of power, performance, cost and area.

- Since PVT variations and other physical implementation issues are beginning to have a major impact on communication architecture selection in UDSM technologies, automated tools for physical implementation aware communication architecture design will be required. Our work on physically aware bus architecture synthesis can be extended by adding PVT variation awareness, to create a more comprehensive synthesis methodology.

- Memory is becoming dominant in emerging MPSoC designs, taking up a majority of the chip area and dissipating significant amounts of power as well as affecting performance, similar to communication architectures. In this thesis we focused on coarse grained memory-communication architecture co-synthesis. This work can be extended by incorporating a finer grained memory synthesis that considers source code optimization (e.g., loop unrolling and array reordering) for better data allocation, and considering the problem of memory hierarchy synthesis.

- Finally, a very exciting area of future research is the exploration of novel materials and paradigms that are not limited to traditional copper wire based communication architectures. A lot of research is needed in the areas of optical, wireless and carbon nanotube based on-chip communication architectures, to determine which (if any) of these disruptive technologies has the potential to
replace copper wires in future MPSoC designs. Already, preliminary experiments on all of these novel interconnect technologies have shown promising results [118].
Bibliography


198


