APPLICATION MAPPING AND NOC CONFIGURATION USING HYBRID PARTICLE SWARM OPTIMIZATION

by

Muhammad Obaidullah

B.Sc. Electrical Engineering, Abu Dhabi University, U.A.E, 2014

A thesis presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Applied Science

in the program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2017

Application Mapping and NoC Configuration

Using Hybrid Particle Swarm Optimization

Master of Applied Science, 2017

Muhammad Obaidullah

Electrical and Computer Engineering

Ryerson University

Abstract

Network-on-Chip (NoC) has been proposed as an interconnection framework for connecting large number of cores for a System-on-Chip (SoC). Assuming a mesh-based NoC, we investigate application mapping and NoC configuration optimization using a hybrid optimization scheme. Out technique, Hybrid Discrete Particle Swarm Optimization (HDPSO), combines Tabu-search, communication volume based core swapping, and swarm intelligence. We employ a Tabu-list to discourage swarm particles to re-visit the explored search space and propose an alternative route towards the intended movement direction. In each iteration of swarm, a sub-swarm containing configuration solutions (sub-particles) searches for optimal configuration for the parent particle (mapping solution). Optimization goals include minimum average communication latency, power, area, credit loop latency, and maximum average link duty factor. The proposed technique is tested for well-known multimedia application core graphs and several large synthetic cores-graphs. It was found that on average our hybrid scheme generates high quality NoC mapping and configuration solutions when compared to some existing stochastic optimization techniques.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Dr. Gul N. Khan. His guidance helped me during the time of research and writing of this thesis. It was fantastic to have the opportunity to work with such a supporting, cheering, and experienced person. He paved my way throughout my study and helped me make crucial decisions in my academic life. He consistently allowed my work to be my own work, but steered me in the right direction whenever he thought I needed it. I could not have imagined having a better advisor and mentor for my MASc study. It has been a period of intense learning for me, not only in the scientific arena, but also on a personal level.

A very special gratitude goes to Ryerson University, Faculty of Engineering & Architectural Science (FEAS), and Ontario Graduate Scholarship (OGS) for helping and providing the funding for the work.

I would also like to thank my siblings: Mamuna, Abdur Rab, Fatima, and Aisha, who have provided me moral and emotional support in my life. I am grateful to my father Rashid Manzur, for supporting me spiritually throughout writing this thesis and my life in general. I am also grateful to my other family members and friends who have supported me along the way.

Last but not the least, I would like to thank my life-coach, my mother Kausar Rashid. She was always there for me. She taught me never to give up. No matter how hard the journey became, she encouraged me to try harder and keep moving forward.

Thanks for all your encouragement! This would not have been possible without you.

Muhammad Obaidullah

Dedicated to

my mother

Kausar Rashid

Table of Contents

Author's Declaration ……………………….………………..…..……………………..… ii

Abstract …………………….……………………………………..……………………… iii

Acknowledgements ……………………………………………..……………………… iv

Dedication ….…………………………..……………………..………………………….. v

List of Tables ………………………………………………………………………… xi

List of Figures…..………………………..……………………..……………………..… xii

Chapter 1 Introduction... 1

1.1 Motivation.. 4

1.2 Objectives & Contributions 5

1.3 Thesis Structure 5

Chapter 2 Overview 7

2.1 NoC Topologies.. 7

2.2 NoC Packet Structure... 11

2.2.1 Flow Control Unit (flit) 12

2.2.2 Physical Transfer Unit (phit) 12

2.3 Flow Control.. 13

2.3.1 STALL-GO Flow Control 13

2.3.2 T-Error Flow Control.. 14

2.3.3 ACK/NACK Flow Control 14

2.3.4 Credit-based Flow Control 15

2.4 Routing Mechanisms 15

2.4.1 Starvation 15

2.4.2 Dead-locks 16

2.4.3 Live-locks 16

2.4.4 Deterministic Routing 16

2.4.4.1 Dimension ordered routing (X-Y) or (Y-X) 16

2.4.4.2 Destination-Tag Routing 17

2.4.5 Oblivious Routing 18

2.4.6 Adaptive Routing 19

2.4.6.1 Load Balanced Adaptive Routing 19

2.4.6.2 Fully Adaptive Routing 19

2.4.6.3 Minimal Adaptive Routing 20

2.5 NoC router micro-architecture 20

2.5.1 Crossbar 21

2.5.2 Input port 21

2.5.3 Arbiter 22

2.6 NoC Mapping 22

2.6.1 State-of-the-Art Mapping tools 25

2.6.1.1 Deterministic Techniques 25

2.6.1.2 Stochastic Techniques 26

2.7 NoC Configuration 28

2.7.1 State-of-the-art NoC Configuration Tools and Work-flows 28

Chapter 3 Optimal NoC Mapping using HDPSO 31

3.1 Introduction 31

3.2 Problem Definition 34

3.3 HDPSO Algorithm 35

3.3.1 A HDPSO Particle 35

3.3.2 Initial Population Generation 38

3.3.3 Particle Movement (Applying Swap Moves) 39

3.3.4 Tabu-List and Particle Reflection 41

3.3.5 Communication Volume based Optimization 45

3.3.6 Evolution of Particles 46

3.3.6.1 Inertia Stage (Stage A) 47

3.3.6.2 Local Convergence Stage (Stage B) 47

3.3.6.3 Swarm Convergence Stage (Stage C) 48

3.3.6.4 Traffic Load Optimization Stage (Stage D) 48

3.3.7 Stopping Criteria 48

3.3.8 Dynamic Tuning Parameters 50

3.3.8.1 Inverse exponential self-confidence and exponential swarm confidence 51

3.3.8.2 Inverse exponential self-confidence and remaining swarm confidence 52

3.3.9 Particle Cost Pre-calculation 52

3.4 Experimental Results for NoC Mapping (Particles) 53

3.5 Summary 63

Chapter 4 Optimal NoC Configuration using Sub-swarm 64

4.1 Introduction 64

4.2 Problem Definition 69

4.2.1 Constraints 73

4.3 Configuration Solution Space 73

4.4 Router Sub-component Library Representation 75

4.4.1 Library 76

4.4.2 Component 76

4.4.3 Configuration 77

4.4.4 Design 77

4.4.5 Property 78

4.5 A Sub-particle (Configuration Solution) 78

4.6 Configuration for Topology 79

4.7 Initial Configuration Heuristics 81

4.8 Sub-Swarm Movement 82

4.8.1 Mapping Edges to Links 84

4.8.2 Buffer Size Optimization 85

4.8.2.1 Conventional Virtual Channel (CVC) 85

4.8.2.2 Dynamically Allocated Multi-Queues (DAMQ) 86

4.9 Convergence and Stopping Criteria 87

4.10 NoC Configuration Results 87

4.10.1 Input Library 88

4.10.1.1 Crossbar Library 88

4.10.1.2 Arbiter Library 88

4.10.1.3 Input Port Library 89

4.11 Summary 98

Chapter 5 Conclusion 99

References 101

Glossary 106

List of Tables

Table 2.1 Flow control protocols at a glance 13

Table 3.1 Application core-graphs used as benchmark 54

Table 3.2 Execution Time Comparison of Benchmark CoreGraphs 57

Table 3.3 Traditional DPSO vs HDPSO results for 32, 64, and 128 core synthetic application core-graphs 61

Table 4.1 Negative Universal IDs (uIds) and their meanings. 78

Table 4.2 Crossbar input library properties 88

Table 4.3 Arbiter input library properties 88

Table 4.4 LLD input port library properties 89

Table 4.5 ViChaR input port library properties 90

Table 4.6 EDVC-FRW input port library properties 91

Table 4.7 EDVC-FW input port library properties 92

Table 4.8 Area/Power for PIP and MPEG4 in optimal, generous, and minimal configurations 93

List of Figures

Figure 1.1 (a) Bus-based interconnection architecture (b) network-based on-chip interconnection. 3

Figure 2.1 (a) Ring (b) Butterfly (c) Hypercube (d) Mesh (e) Torus (f) Binary Tree (g) Custom Mesh-Type (h) Star (i) Fully-Connected Topology. 10

Figure 2.2 (a) A 3 x 3 2D mesh NoC (b) A 3 x 3 torus NoC. 11

Figure 2.3 Packet, flit, and phit relationship. 12

Figure 2.4 Stall-Go flow control. 14

Figure 2.5 T-Error flow control. 14

Figure 2.6 Ack/Nack flow control. 15

Figure 2.7 Clockwise and Anti-clockwise dead-lock situations. 16

Figure 2.8 X-Y routing mechanism for 3 different source and destination packets. 17

Figure 2.9 Routing in 4-ary 2-fly butterfly NoC topology 18

Figure 2.10 Oblivious routing mechanism. 18

Figure 2.11 NoC router micro-architecture. 21

Figure 2.12 NoC design workflow and this work’s targeted workflow. 23

Figure 2.13 Mapping possibilities for a 2 by 2 mesh NoC with 4 cores. 24

Figure 2.14 Black box analogy of NoC mapping tool. 25

Figure 2.15 Æthereal NoC configuration and mapping workflow. 29

Figure 3.1 DPSO Search spaces – particles reach sub-optimal solutions 32

Figure 3.2 HDPSO Algorithm. 32

Figure 3.3 Application with multiple modes of operation (AM). 35

Figure 3.4 A HDPSO particle structure for MPEG-4 decoder. 37

Figure 3.5 Determining distance between particles (particle A & B). 39

Figure 3.6 Identifying the chosen swap move. 40

Figure 3.7 Complex swap moves. 41

Figure 3.8 Reflection of particles from the explored search space. 42

Figure 3.9 HDPSO vs DPSO particle cost variations for mapping of MPEG4 decoder. 43

Figure 3.10 Reflection of a Tabu-ed swap (2,4) move. 44

Figure 3.11 Reflection of a swap (2, 4) with respect to y-axis. 44

Figure 3.12 Communication Volume based core swapping. 46

Figure 3.13 Local Convergence Stage (Stage B) path decider. 47

Figure 3.14 Global cost for same confidence - DPSO *vs.* HDPSO. 50

Figure 3.15 Inverse exponential self-confidence and exponential swarm-confidence. 52

Figure 3.16 Inverse exponential self-confidence and remaining swarm-confidence. 52

Figure 3.17 Particle cost pre-calculation. 53

Figure 3.18 Communication cost normalized to NMAP cost. 54

Figure 3.19 Cost improvement DPSO *vs.* HDPSO. 56

Figure 3.20 Traffic patterns used to evaluate the HDPSO based NoC mapping (a) Bit-reversal, (b) Neighbor (c) Tornado (d) Transpose (e) Shuffle (f) Stencil. 59

Figure 3.21 HDPSO generated optimal (a) 3×4 (b) 4×4 NoC mapping of MPEG4 decoder. 62

Figure 3.22 HDPSO generated optimal (a) 3×4 (b) 4×4 NoC mapping of VOP decoder. 62

Figure 4.1 CAD system architecture for NoC synthesis 65

Figure 4.2 Typical NoC synthesis workflow. 66

Figure 4.3 (a) Segregated search space optimization workflow (b) combined search space optimization. 67

Figure 4.4 The proposed hybrid workflow for NoC mapping and configuration. 68

Figure 4.5 Inner workings of the mapping and configuration HDPSO algorithm. 69

Figure 4.6 (a) Average duty factor and buffer penalty relationship (b) Average duty factor and configuration cost relationship. 73

Figure 4.7 NoC Configuration solution space. 74

Figure 4.8 NoC Configuration library XML element hierarchy. 75

Figure 4.9 Library element hierarchy and relationship. 75

Figure 4.10 A library element defined in XML format. 76

Figure 4.11 A component element defined in XML format. 77

Figure 4.12 Internal structure of a sub-particle. 79

Figure 4.13 Identification of un-used input ports 80

Figure 4.14 Determining distance between sub-particles (configuration solutions). 81

Figure 4.15 Initial Configuration heuristics. 82

Figure 4.16 HDPSO sub-particle movement algorithm. 83

Figure 4.17 Link traffic volume calculation. 84

Figure 4.18 Link stress estimation. 84

Figure 4.19 CVC input port architecture. 85

Figure 4.20 Determining minimum number of buffers required. 86

Figure 4.21 DAMQ VC input port architecture. 87

Figure 4.22 Power and area % savings observed when comparing HDPSO optimized configuration to un-optimized generous configuration. 94

Figure 4.23 NoC coregraphs (a) MPEG4 (b) 16-core VOPD (c) DVOPD (d) PIP. 95

Figure 4.24 NoC configuration and mapping solutions (a) MPEG4 (b) VOPD (c) DVOPD (d) PIP. 96

Figure 4.25 Cost improvements due to configuration optimization. 98

Chapter 1

Introduction

To meet complex functions support, minimum area, and small power consumption demands of embedded devices, System-on-a-Chip (SoC) designers are squeezing multiple and more complex Silicon Intellectual Properties (SIPs) onto a single chip. However, major design challenges while integrating multiple SIPs include reducing chip area and power consumption while maintaining high performance specifications. To deal with interconnection complexity, abstraction and regularity of interconnection architecture is required. Network-on-Chips (NoCs) have emerged recently as viable interconnection alternative instead of buses. NoCs provide high data bandwidth, high network throughput, multiple transactions per cycle, and scalability.

Traditionally, SIPs or cores were mapped using static bus architectures eg. AMBA, AXI, STBus etc. A conventional non-hierarchical bus architecture is shown in Figure 1.1(a). Slaves (S) are SIPs which respond to requests or commands of Masters (M) SIPs on the bus. Since there are multiple masters, bus access is controlled through a component called arbiter (A). Usually, bus masters are Central Processing Units (CPUs) and Direct Memory Accessors (DMAs) while slaves are peripherals, I/Os, and memories.

Bus architecture has many drawbacks when increasing the number of SIP blocks on the SoC. Since bus is a collection of wires, only one core can use those wires to communicate to another core. The core which initiates the signals and generates control signals is called a bus master. At a time, only one master and one slave can communicate with each other. This is to ensure that there are no short-circuits because one core has pulled the signal low while the other core pulls it high. While two cores are communicating with each other, other cores might also be waiting for bus access and precious time is being wasted.

Secondly, every SIP block attached in bus adds a parasitic capacitance degrading the data signal integrity. This is a huge issue in bus architectures that do not use tri-state buffers to disconnect cores which are not communicating. For example, Avalon bus architecture uses multiplexers as switching component rather than tri-state buffers. This increases parasitic capacitance as more SIPs are added onto the bus.

Thirdly, data bandwidth in bus-based systems is limited and is shared among all IPs attached because only one transaction can occur in a single clock cycle. If there is contention for bus-access, lower priority masters are forced to wait until higher priority masters have completed their transaction. This is true even if the master A wants to talk to slave C and master B wants to talk to slave D. This can cause a starvation condition in which masters with low priority never get access to resources. To tackle this issue, SoC designers are forced design and install custom hardware to detect starvation condition.

Lastly, bus arbitration logic delay grows as more number of masters are added onto bus. [1] The decision tree of who should get the bus access grows as more bus nodes are added and so does the arbitration logic, area occupancy, and power consumption. This large arbitration ladder logic increases critical path delay which in turn reduces the maximum operational frequency and degrades the performance.

A typical on-chip network architecture is shown in Figure 1.1(b). Label ‘C’ signifies SIP **C**ores and are connected to other SIPs through **R**outers labelled ‘R’. NoC allows core’s signals to be packetized and transmitted over fabric of switches and routers. Packetization of data also allows each channel’s bandwidth to be shared among multiple sources using Virtual Channels (VCs).

NoCs improve throughput, are easily scalable, and allow multiple transactions to occur in a clock cycle. However, NoCs still need to overcome several challenges such as significant static and dynamic power consumption, large chip area occupied by network hardware, traffic congestion, and high data transfer latencies. Network resources also need to be managed efficiently in order to avoid expensive design costs while keeping the networks performance maximum.

There are three NoC optimization metrics we will be addressing in this work, average communication latency, power, and area. We will be using a library of pre-synthesized NoC router components and information about application (CoreGraph) to optimize these metrics. The architecture of these pre-synthesized router components are discussed by Oveis-Gharan and Khan [2, 3, 4, 5, 6, 7]. These pre-synthesized components are imported into our tool and their power and area metrics are presented in Section 4.10.1. Our optimization method is a hybrid of well-known stochastic techniques Discrete Particle Swarm Optimization (DPSO) [8] and Tabu-Search (TS) [9]. However, we have modified the underlying DPSO algorithm significantly to cater to our NoC mapping and configuration solution space. DPSO algorithm will be discussed in detail in chapter 3.

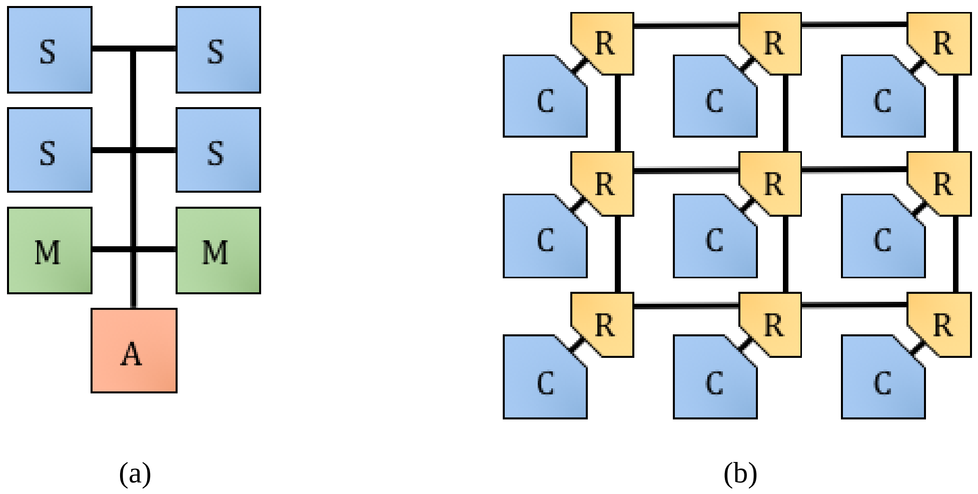


Figure 1.1 (a) Bus-based interconnection architecture (b) network-based on-chip interconnection.

Power consumption is composed of two sub-components, static and dynamic. Static (leakage) power consumption is mainly due to reverse-bias leakage between diffused regions and the substrate of transistors. This power consumption is due to the small current leaking through a transistor even-though depletion region is thick and present. Useful analogy is to imagine a water tap which was left little open and water droplets are still falling. Like in the water tap, a lot of force is required to close the tap completely, high voltage needs to be applied to the gate of the transistor to close it completely. However, this high voltage is not available to the transistor (VSS and VDD) because modern chips and cores operate at around 1.1V or even lower subject to the operating frequency requirement and the technology cell library specification.

Dynamic (switching) power consumption is calculated by adding transient power and capacitive-load power consumption. Dynamic power consumption is due to the current that flows only when the transistors of the devices are switching from one logic state to another. This power is consumed in charging external load capacitance and is dependent on switching frequency. Dynamic power consumption increases when more data transfers or transactions occur among network nodes/routers. One way to reduce dynamic power consumption in NoCs is to place SIPs which communicate with high volume near each other at minimum hop distance. In this work, we use the worst-case dynamic power for each router component. This is calculated by simulating sending of toggling bits through the input ports of the component.

Flit buffers are major contributors to network static power consumption [10]. Flit buffers are essentially Static Random Access Memories (SRAMs) used to store small data units called f**l**ow control un**its** (flits) in the router input ports. As technology process nears submicron level, static power consumption gains major share in total power consumption [10]. SRAMs are major shareholders of chip area because 1 D-flip-flop holds 1 bit and we need 8 of those to store just 1 byte. For example, if the flit size is 128-bits and the buffer depth is 4 flits for each input port, a 4x4 mesh NoC will have total $128\times4\times5\times16 = 40{,}960$ flops. On top of that, all modern ASICs have Memory Built-In Self-Test (MBIST) components inserted for each D-flip-flop. This further increase the chip area requirement even-though MBIST hardware is redundant in normal functional mode of the chip. Hence, it is becoming crucial to choose right amount of flit buffers in an on-chip network to reduce NoC power and area.

1.1 Motivation

Three major VLSI hardware trends compel us to do research on NoC optimization tools. The first is increased usage of NoCs on SoCs and microcontrollers for IoT. SoC designers are realizing that there is simply no better way to integrate large number of cores onto a single chip rather than a NoC. Each core's designer can request a throughput from SoC team and be assured that the throughput is going to be available no matter how many cores are being added on the chip level. Because each additional core brings with itself more network throughput, bandwidth is not shared, unless two or more cores are trying to communicate with the same core. Yesterday's researches often predicted that NoC usage will become a reality in future. The future is here and it is looking bright for NoCs. The second is lack of optimization tools that optimize for heterogeneous SoCs which are often required to operate on wide variety of operating modes involving multiple cores on a chip. There is a need for optimization tool which co-optimizes NoC mapping and NoC configuration, takes into consideration multi-modal application, and uses a stochastic optimization technique. The third reason is that most of the NoC optimization tools do not use pre-synthesized router components to optimize NoC latency, power, and area. NoC area and power can be more accurately measured once the values for real synthesized components is available. Existing tools often try to use parametric equations to predict the area and power of a NoC before synthesis and those can sometimes be far-off from real values. Due to these reasons, we have proposed a C++ based tool which performs optimization of NoC area, power, and latency by using library of pre-synthesized router components.

1.2 Objectives & Contributions

Our main contributions include combining exploration abilities of particle swarm with exploitation abilities of traffic-load directed swapping (Section 3.3.6.4), development of linear particle deflection mechanism (Section 3.3.4) based on Tabu memory which is well-suited for NoC mapping problem space, combining particles and sub-particles to form con-joined search space for mapping and synthesis problems (Section 4.8), defining workflow for NoC mapping and configuration optimization (Section 4.1), and proposing a stopping criteria based on the average mapping solution distances and swarm divergence factor (Section 3.3.7). All in all, the resulting optimization tool is based on Hybrid Discrete Particle Swarm Optimization (HDPSO) method and is tested for some synthetic and real-life applications yielding promising results (Section 4.10).

1.3 Thesis Structure

The remaining thesis is organized as follows. Chapter 2 provides an introductory background on on-chip networks. We introduce some basic NoC topologies and how their routing mechanisms work. Secondly, NoC Packet structure and micro-architecture of a typical NoC router is discussed. Thirdly, past works in NoC mapping and configuration domain will be discussed.

In Chapter 3, we define NoC mapping, which is the first half of optimization problem. We also discuss how an application is parsed into CoreGraph suitable for input into our optimization tool. We describe the inner workings and algorithm of particle solution space. We also discuss the tool turning parameters and propose optimization stopping criteria. We conclude the chapter with experiment on synthetic and real applications and present results from the tool while comparing it against traditional DPSO and other methods.

In Chapter 4, we characterize NoC configuration problem and propose a sub-swarm technique to solve it. We also propose design constraints which allows the tool to restrict to interesting router configuration solutions. Additionally, we also list and detail NoC component libraries we used for optimization. We compare our sub-swarm with un-optimized configuration results and provide the improvement metrics achieved. Chapter 5 concludes the thesis and provides future direction for future work.

Chapter 2

Overview

2.1 NoC Topologies

NoC allows SIPs or modules to be connected in any topology with aid of on-chip routers. Popular on-chip network topologies are 2D Mesh, Hypercube, Torus, and Butterfly. An n-dimensional radix-k torus, or k-ary n-cube, consists of $N = k^n$ nodes arranged in an n-dimensional cube with k nodes along each dimension. Each node in the topology is assigned an n-digit radix-k address $\{a_{n-1}, \dots, a_0\}$ and is connected by a pair of channels (one in each direction) to all nodes with addresses that differ by $\pm 1(mod\ k)$ in exactly one of the address digit [11].

Popular NoC topologies being used by the researchers include ring, butterfly, hypercube, mesh, torus, tree, custom-mesh, star, and fully-connected as shown in Figure 2.1. In figure 2.1, ‘R’ signifies a Router (node with a local core attached), and ‘S’ indicates a switch (a node with no core attached to itself). We consider a 2-D mesh topology for this work and modify it according to the application needs such that it ends up becoming a custom-mesh type topology Figure 2.1g after optimization.

A ring topology (Figure 2.1a) has advantages of containing very few links and 2 data paths to a specific router i.e. clock-wise and anti-clock wise. However, each router is connected to only 2 other routers. Therefore, most of the remaining cores are at greater than 1 hop distance away. A ring topology is rarely used in NoC because of low radix, the fact that wires are cheap (in terms of area and power) on the chip, and high hop-count and latency for routers which are on the other side of the ring.

A butterfly topology (Figure 2.1b) has frequently communicating cores placed near each other while cores that communicate less frequently are placed at more hop distance. The Figure 2.1b shows a folded perspective of butterfly topology. However, butterfly topology can have very congested routers which can increase latency of the network and is difficult to scale because addition of each core onto the network not only requires addition of a router (as with all topologies), it requires addition of more switches.

Hypercube topology (Figure 2.1c) is a very interesting topology for 3D Integrated Circuits (ICs) as it allows components on the top and bottom silicon layers to be incorporated into the mesh network. However, this means that each router will have 2 additional input and output ports compared to a mesh topology which increases the complexity of the network. In case of 2D ICs, this topology increases the interconnection complexity posing difficulty in floor-planning and increases router size (increased power and area consumption) while not presenting significant trade-off advantages in network latency.

2-D Mesh network topology (Figure 2.1d) has regularity of design, increased network throughput with addition of any new node, and alternative routes to destinations providing fault-tolerance A 2D-mesh network has a router connected to each core and each router connected to four other routers..

2D-torus network topology (Figure 2.1e) has the same advantages as a mesh network and some more. A 2D-mesh network is often congested in the center because the radix of the boundary routers is low, radix 2 for corners and radix 3 for routers that are at boundary but not corners. The torus topology removed this problem by making the topology circularly symmetrical which increases the radix of boundary routers and increases possible data paths. Disadvantages include increased chip area and power compared to mesh network due to addition of input and output ports on the boundary routers.

Tree topology (Figure 2.1f) is good for small applications which have groups of cores which communicate among themselves in high volume (Locality). Tree topology can take advantage of locality and place highly communicating cores under the same tree node while less frequently communicating cores in different branches. As the application size grows, tree structure can grow exponentially as addition of 1 core can cause root structure to grow too in order meet the increased throughput demand. However, this topology is widely in use in todays SoCs because it supports hierarchical approach which block designers, core designers, and SoC integration designers take while designing ASICs. The block-level designers can design a router for their internal sub-components and have one link installed in it for communicating with the upper core level logic. The core-level designers may design a router for their internal blocks and have one link installed in it for communicating with the upper SoC level logic.

For an application specific mesh topology when a specific static routing mechanism (eg. XY-routing) is chosen and fixed, there tends to be some links in the mesh network which never get used. So, the chip designers can remove the links and the associated hardware to reduce area occupied by the NoC and NoC power consumption. What results after this, is a custom mesh-like topology (Figure 2.1g). In this topology all links which never get used are removed and the routers area and power is greatly reduced. This topology is very interesting for applications containing heterogeneous cores. Unless the design contains N number of processing units (multicore CPUs) or N number of compute units (SIMD/GPU workgroups), there are always some links in the mesh network that can be trimmed off. Almost all real-word applications contain heterogeneous cores and there is a need to customize a popular homogenous-friendly mesh topology to become heterogeneous.

Every core in a star topology (Figure 2.1h) is at most 2 hop distances away from any other core. However, it is rarely used in a NoC because of poor scalability. Every additional core requires addition of a local router and expanding the central router. Applications with more than 10 cores require a huge central router increasing the chip area and power requirements. Additionally, central router is very congested as all the traffic must pass through it.

Fully connected mesh network (Figure 2.1i) is a perfect topology in terms of latency as the hop distance from any core to another is just 1. However, it is a nightmare for floor planning and does not scale well with application. Each additional core requires addition of another router and modification of all other routers as well (to accommodate one more input and output port).

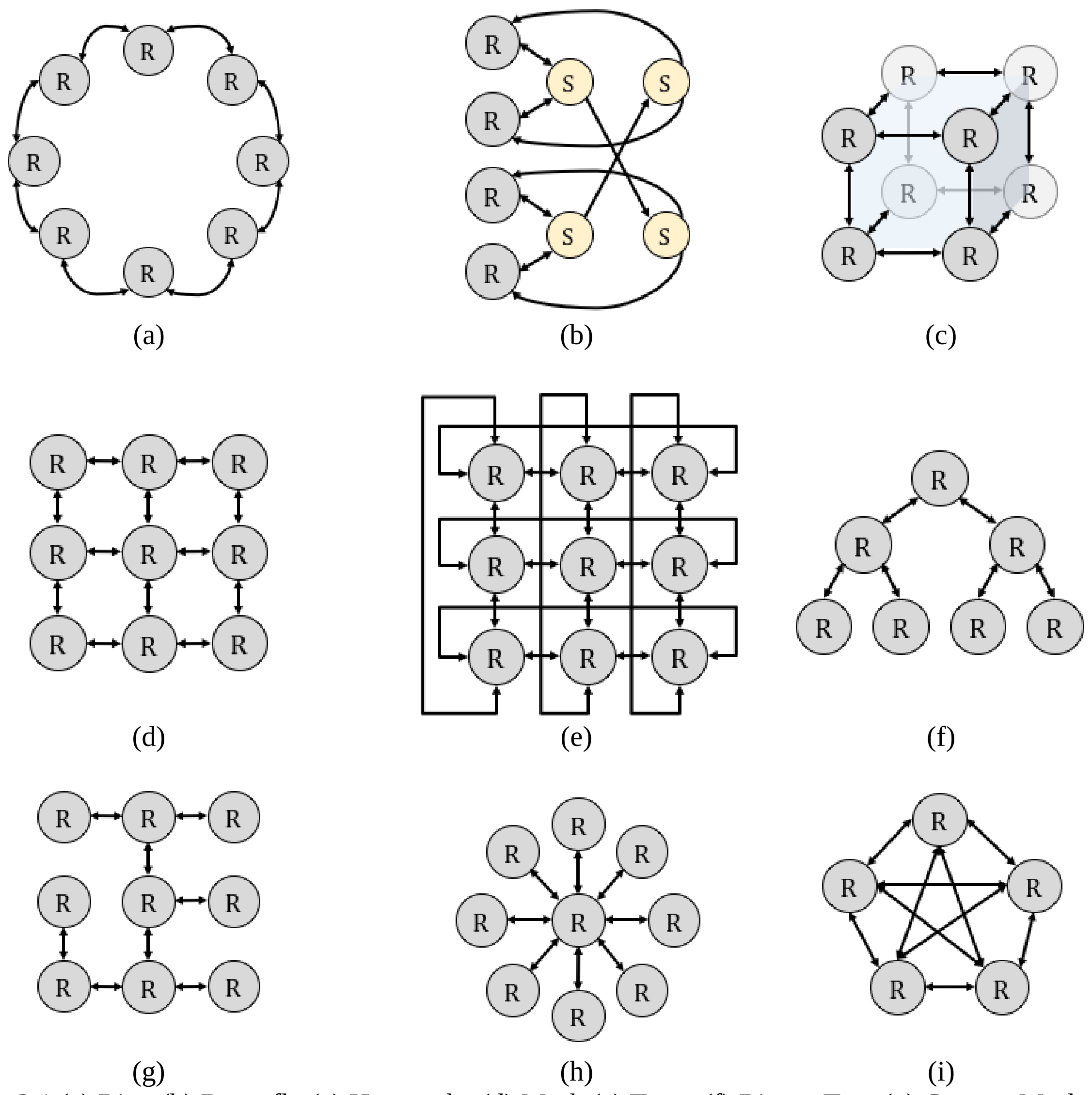
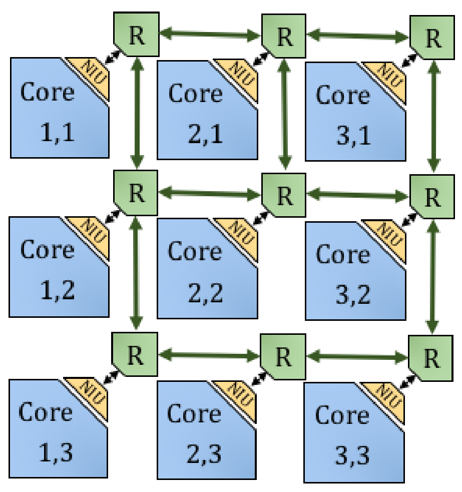


Figure 2.1 (a) Ring (b) Butterfly (c) Hypercube (d) Mesh (e) Torus (f) Binary Tree (g) Custom Mesh-Type (h) Star (i) Fully-Connected Topology.

On-chip mesh network and torus network are shown in Figure 2.2. Each core is placed at a tile in the topology. A tile is an empty seat or place in the network at a specific location in which a core can be connected. Each core is then connected to its appropriate Network Interface Unit (NIU) and NIUs are connected to routers marked as R in Figure 2.2.



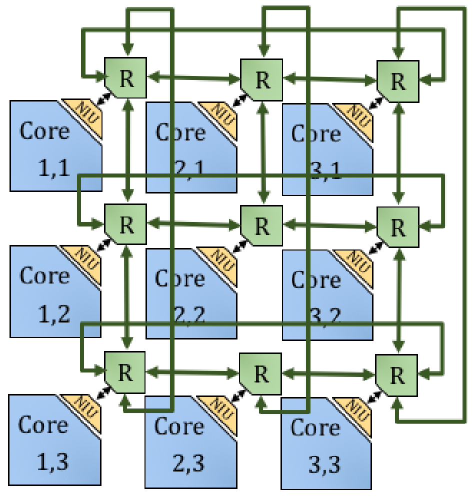


Figure 2.2 (a) A 3 x 3 2D mesh NoC (b) A 3 x 3 torus NoC.

2.2 NoC Packet Structure

Bus transactions signals from SIPs are converted into packets by Network Interface Unit (NIU) before sending it to NoC fabric. NIU is also responsible for setting the priority of the packet and in case of transaction, it makes sure that no other data reaches the core until the transaction is complete. It also deconstructs the in-coming packets and generates appropriate signals for the core to understand. NIUs can be different for each core according to the interface signals they have. For example, a core with Advanced eXtensible Interface (AXI) bus interface will require a NIU that converts AXI bus signals into network packets. Usually theses are standard NIUs and its Register Transfer Level (RTL) hardware can be generated by a synthesis tool easily. A conventional NIU contains Generic Core Interface (GCI), Packet Maker (PM), Packet Disassembler (PD), and Asynchronous FIFOs for buffering and Clock Domain Crossing (CDC) synchronization [12]. Any core specific wrapper can communicate with the GCI with simple predefined handshaking signals to initiate read and write to and from NoC. PM assembles packets for the network and places them in order in the PM memory from where an asynchronous FIFO can take it out and send to the network. PD takes packets from the asynchronous FIFO and decodes them into signals for the GCI. Since on-chip network might be operating at a different clock/frequency than the core, there is need for asynchronous FIFOs to allow different read and different write speed. Synchronization blocks and buffers are added to the control signals to remove CDC and meta-stability issues.

A packet is composed of a header flit, some or no body flits, and a tail flit as shown in Figure 2.3. The header flit contains metadata about the packet. For example, source id, destination id, length of packet, and Virtual Channel (VC) id if VCs are used. Each flit can be broken down into Physical Transfer Units (phits) according to the physical interconnection bus width. Phits consist of bits which are transmitted through the interconnection fabric one clock cycle at a time.

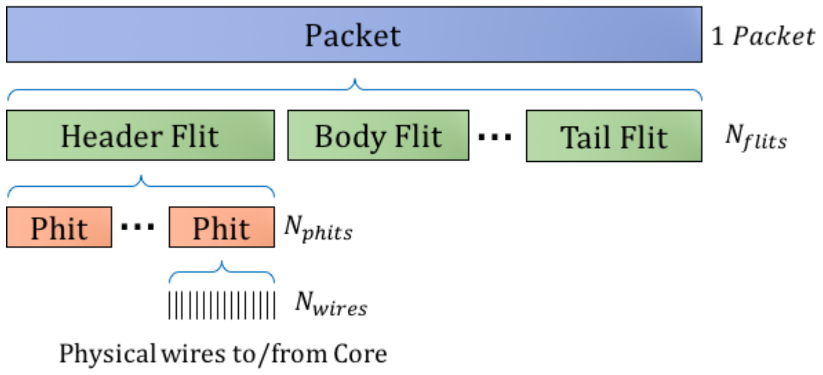


Figure 2.3 Packet, flit, and phit relationship.

2.2.1 Flow Control Unit (flit)

Network packets are broken down into flits so that arbitration and flow control can be done on each flit individually. For example, arbiter generates grant signals for a particular flit, a credit is generated for a particular flit, etc. When a grant signal is received for a flit, all bits in the flit must be transmitted from input port to the output port. Similarly, when an input port’s buffer becomes available, a credit is generated for the upstream router so that it can send another flit. Breaking down packets into flits presents several advantages because flit size is fixed as per the capability of routers to store them. A router can easily store flits in their input port buffers, whereas it is difficult to store the whole packet especially if the length is varying. Since memory is a huge consumer of chip area and power, it is better to keep buffer quantity as low as possible. We will be addressing optimization of these input port buffers in this work (Section 4.8.2).

2.2.2 Physical Transfer Unit (phit)

Phit is composed of bits which are transferred from one router’s output port to another router’s input port in one clock cycle. For those interconnection architectures where a flit size is larger than the phit size, multiple clock cycles are used to transmit one flit and the input port also contains a shift register to move the bits over as soon as next bits arrive. As soon as the shift register is full, the flit is latched into

FIFO and a request might be generated for the arbiter. Making phit size smaller than flit size requires less interconnection wires from the source output port to the destination input port. There is very little advantage in terms of chip area and power savings to reduce number of interconnection wires. Interconnection wires are easy to synthesize, they increase network throughput, and reduce clock cycles required to transmit data. On the other hand, decreasing interconnection wires will provide very little area and power savings. Additionally, if phit size is smaller than flit size, a shift register is required to shift the previous bits over and latch-in the next in-coming bits. Therefore, in this work, we assume flit size is equal to the phit size.

2.3 Flow Control

Flow control in a network basically defines set of rules to move data from the sender to the receiver. It is a protocol using which routers communicate with each other to transfer data. Flow control also involves routers informing other routers of its intentions and status by using meta-data and control links. Table 2.1 shows buffering requirements and features of popular flow control protocols.

Table 2.1 Flow control protocols at a glance

Characteristic	STALL/GO	T-Error	ACK/NACK	CREDIT-BASED
Buffer Area	2N+2	3N+2	3N+k	2N+2
Logic Area	Low	High	Medium	Low
Performance	Good	Good	Depends	Good
Power (est.)	Low	Medium/High	High	Low
Fault Tolerance	Unavailable	Partial	Supported	Unavailable

2.3.1 STALL-GO Flow Control

In this type of flow control, the receiver informs the sender whether it is ready to receive more data. This is a type of explicit flow control where the control decision is taken at the receiver's end.

The router is modified to have two more signal wires other than phit length (Figure 2.4). One signal forward indicating presence of new data and another backward informing condition of input buffers either Filled('STALL') or Empty('GO'). If not enough buffer space is available in the next router/repeater, the stall signal is send to the previous router/repeater.

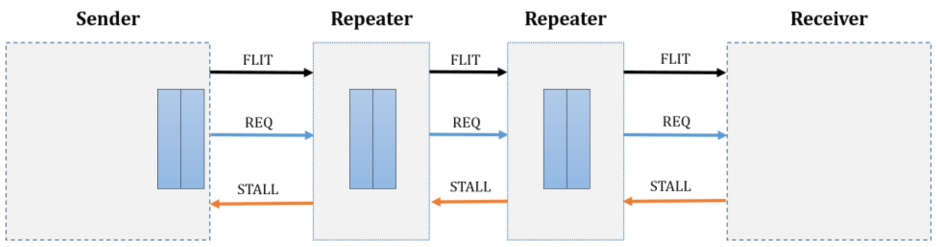


Figure 2.4 Stall-Go flow control.

However, the main disadvantage of STALL/GO flow control is that no fault handling is done should any flit get corrupted. In case of a corrupted flit, the error handling task is delayed over to some higher-level protocol.

2.3.2 T-Error Flow Control

This type of flow control is similar to STALL-GO flow control except that an extra resynchronization stage is added between the end of the link and the receiving switch as shown in Figure 2.5. T-Error flow control often increases the operating frequency of the link. This requires an additional resynchronization stage near the end of the link which is done with the help from combination of clock and delayed clock signals.

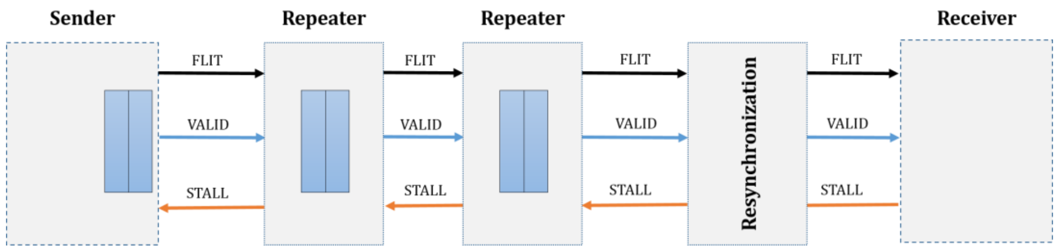


Figure 2.5 T-Error flow control.

2.3.3 ACK/NACK Flow Control

In this type of flow control, a copy of the flit is kept at the sender's end and the flit is sent. If the receiver acknowledges that it received the packet, the flit is discarded and the next one is sent as before procedure. As shown in Figure 2.6, repeaters on the link can be simple registers while the number of buffer requirements for the sender and receiver side is 2N+k buffers to guarantee maximum throughput. N: Number of repeaters.

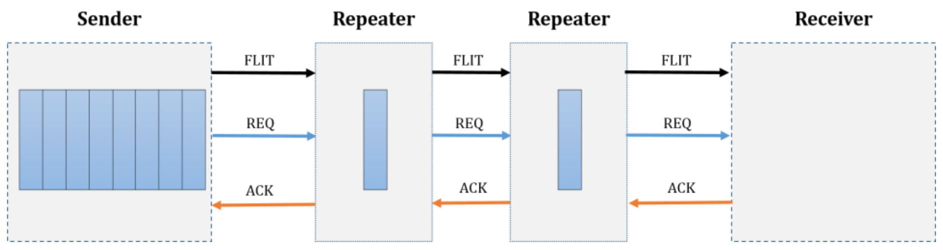


Figure 2.6 Ack/Nack flow control.

It can be implemented in two ways, end-to-end and switch-to-switch. In end-to-end, the copies of flits are kept at the sender side and the final receiver acknowledges reception of flit. On the other hand, in switch-to-switch, the copies of flits are kept in any sending switch and the acknowledgement of flit received is sent back from the next switch.

2.3.4 Credit-based Flow Control

In this type of flow control, the sender keeps a count of the number of free flit buffers in each virtual channel in the receiver. Anytime the sender sends the flit to receiver, the count is decremented and it is incremented if the receiver consumes/forwards the packet further. One credit connection is required from the receiver router to the sender router. When the receiver pulls this signal high, this indicates to the sender that a packet just left receiver’s buffer and there is an empty space for it to send another packet.

2.4 Routing Mechanisms

Routing mechanisms determine how packets/flits move from source/sender to sink/destination. There are basically two categories of routing algorithms. A good routing algorithm must prevent any potential dead-lock, starvation, and live-lock situations of packets/flits in the network.

2.4.1 Starvation

Starvation is denying a core access to necessary resources. Routing starvation problem occurs when the routing algorithm services an area of routers heavily while neglecting another area. A routing algorithm might prefer to service cores on the right side of the mesh. So, the packets travelling east reach first and have higher priority than packets travelling west. This will cause starvation to the cores present on the left side of the mesh.

2.4.2 Dead-locks

Deadlock situation may arise in NoC when any circular waiting path is generated from the routing algorithm. Any waiting for acknowledgement or waiting for credit path which cycles back to the original router is a deadlock. For example, router 1 is waiting for acknowledgement/credit from router 2, router 2 from router 3, router 3 from router 4, and router 4 from router 1 again. Two types of dependency cycles can be formed in a mesh network, clockwise and anti-clockwise; those are shown in Figure 2.7. To prevent any deadlock, at least one of the cyclic dependency must be broken. This is done by either creating virtual channels or by restricting the packet movement in a specific dimension/direction.

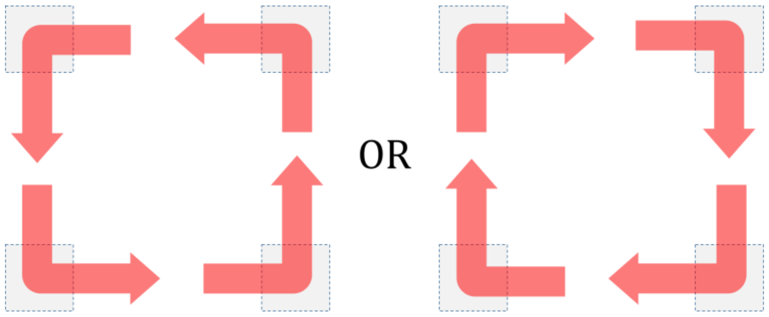


Figure 2.7 Clockwise and Anti-clockwise dead-lock situations.

2.4.3 Live-locks

In this situation, flits keep travelling through the network in circle without reaching a destination. Because the flits do move instead of being stored in a buffer and not transmitted, as in dead-lock situation, we call it 'live'-lock. Live-locks mainly occur when deterministic routing is not used or flits are allowed to take non-minimal paths to the destination.

2.4.4 Deterministic Routing

The route taken by the packet in reaching from source/sender to sink/destination is pre-determined and known. Each source and sink pair have a unique route which is constant throughout the run time.

2.4.4.1 Dimension ordered routing (X-Y) or (Y-X)

The flit first moves in one dimension and reaches the destination's row or column and then continues to move in other dimension to reach destination. In this type of routing, each router knows where other routers are and can determine if the flit's destination lies at west, east, south or north of it. An example of X-Y dimension ordered routing is shown in Figure 2.8. Each flit first decides it needs to go east or west to reach the column of destination. Then it travels north or south to reach the row of destination. This way a packet which has travelled north or south, never takes east or west turn. Two turns become illegal to break the dead-lock situation. A packet travelling north/south will never take east turn and a packet travelling north/south will never take a west turn.

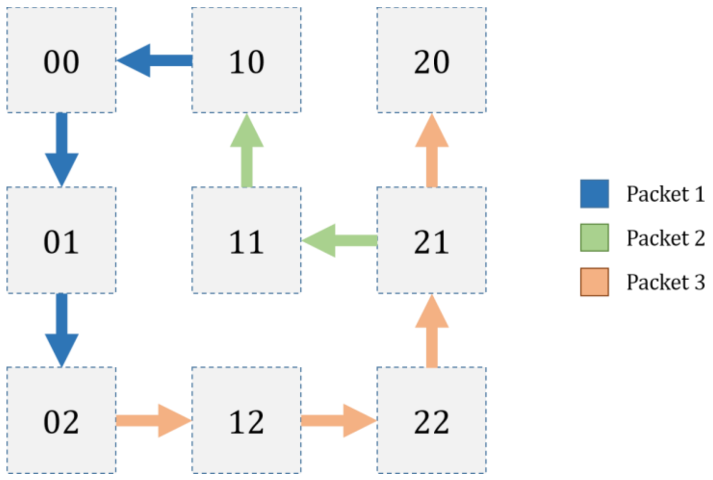


Figure 2.8 X-Y routing mechanism for 3 different source and destination packets.

2.4.4.2 Destination-Tag Routing

Each packet from the sender/source to the receiver/sink is tagged with a destination address in the header. Based on this tag, the intermediate nodes/routers determine the path packet has to take using the routing table. An example of this type of routing is a 4-ary 2-fly butterfly network shown in Figure 2.9.

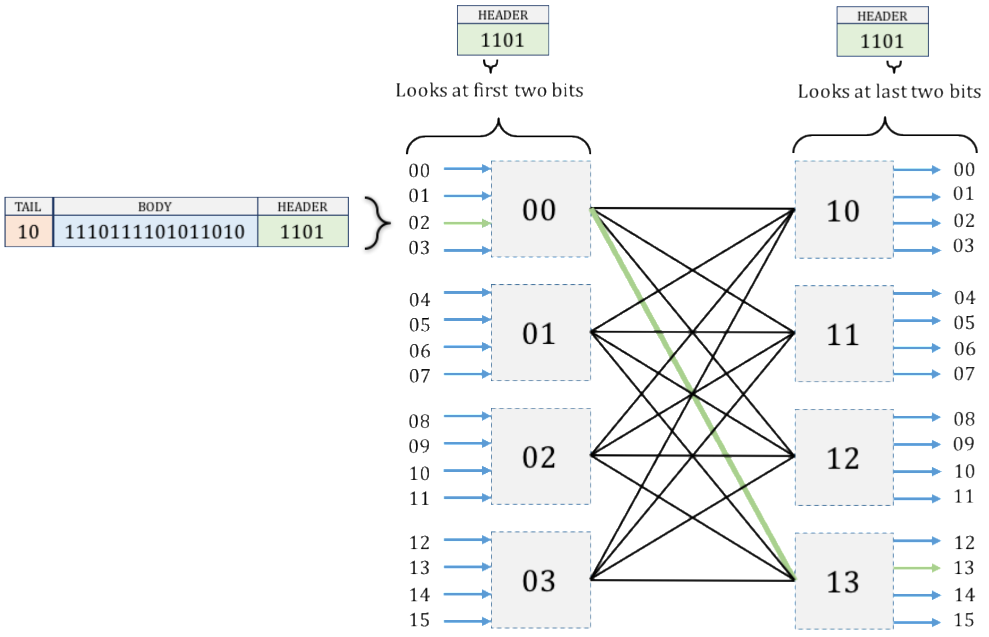


Figure 2.9 Routing in 4-ary 2-fly butterfly NoC topology.

2.4.5 Oblivious Routing

Oblivious literally means being "unaware of". In this type of routing, a packet is routed without regard for the current state of the network. If a packet is to be send from A to B, another node is chosen at random by the sender A and the packet is sent first to this randomly chosen node. Then this node forwards the packet to node B. The Figure 2.10 shows the process of oblivious routing.

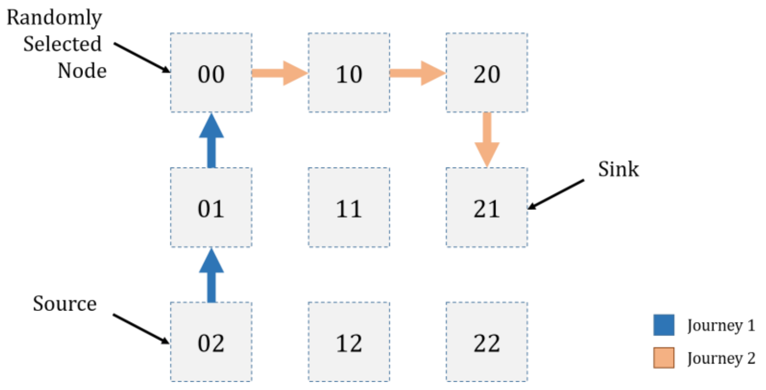


Figure 2.10 Oblivious routing mechanism.

This type of routing causes the load on all channels to distribute and spread out toward the network rather than sticking to one side of the network. The advantage of oblivious routing is that the channel loads and traffic patterns are linearly related. Hence it makes it easy to compute ideal throughput and traffic pattern from given channel load. [13]

However, because the source router can select any intermediate router to send the message first to, the traffic is widespread over the whole network. To preserve locality of the network, the intermediate router is chosen from a defined region around the sender with minimal path. This known as minimal oblivious routing.

2.4.6 Adaptive Routing

Adaptive routing algorithms use information about the network in real-time to adapt and change how different packets are routed on the network. The major challenge in adaptive routing is the fact that the route for each packet is decided in real-time just before a packet is sent from one node to another which can cause live-locks. When a router receives a packet, it calculates the path to destination according to the algorithm hardware.

2.4.6.1 Load Balanced Adaptive Routing

The algorithm tries to equalize the traffic load on all the links equally. An example of load balanced adaptive routing is given in [14] where the authors try to balance the load by using Ant Colony Optimization (ACO). ACO is a biologically inspired from ant colonies and how they locate optimal paths by use of pheromones, attractants, and accumulation of paths. They propose a NoC version of ACO which was used extensively in Wide Area Networks and call it Regional ACO-based Cascaded Adaptive Routing (RACO-CAR). This technique eliminates tables which contain redundant information, shares routing tables with neighboring routers and merge information, and cascades routing to distribute the load in the network.

2.4.6.2 Fully Adaptive Routing

This type of routing algorithm adapts to the congestion and blocking in the network and re-routes the packets away from congested or blocked areas. Liu *et al.* propose a fully adaptive routing algorithm called FreeRider [15]. A non-local adaptive congestion aware algorithm where the congestion information of whole network (non-local) is used to make routing decisions. Rather than using the Congestion Propagation Network (CPN) to propagate congestion information, but instead they propose to use free bits in header flit to carry congestion information. This improves the throughput, shortens the latency, and reduces the power consumption.

2.4.6.3 Minimal Adaptive Routing

Minimal route searching algorithms look for the shortest/minimal path to the destination before sending each packet. Since it is a type of adaptive routing, this algorithm looks for congestion and prefers another route if it is of same hop distance. However, NIUs have to take care of out-of-order arrival of flits.

2.5 NoC router micro-architecture

A typical NoC router is composed of input ports, output ports, an arbiter, and a switch [16] shown in Figure 2.11. Minimum delay of the data path determines a maximum data frequency of the router. An arbiter delay dominates among the other critical path delays and determines the maximum frequency of the router (F_{max}) [2]. When designing a router, minimization of latency while meeting bandwidth requirements is of utmost importance. Extensive amount of work has already been done in developing power and area efficient designs of arbiters, crossbars, and input ports [7, 4, 3, 2, 17, 5, 6, 16]. NoC router buffers are expensive in terms of power and area and constitute as fundamental variable in NoC optimization [18]. Oveis-Gharan and Khan have extensively compared existing router architectures with special focus on their EDVC organization [2]. We propose an algorithm to determine the optimal router components which results in an optimal performance NoC with lower area and power costs.

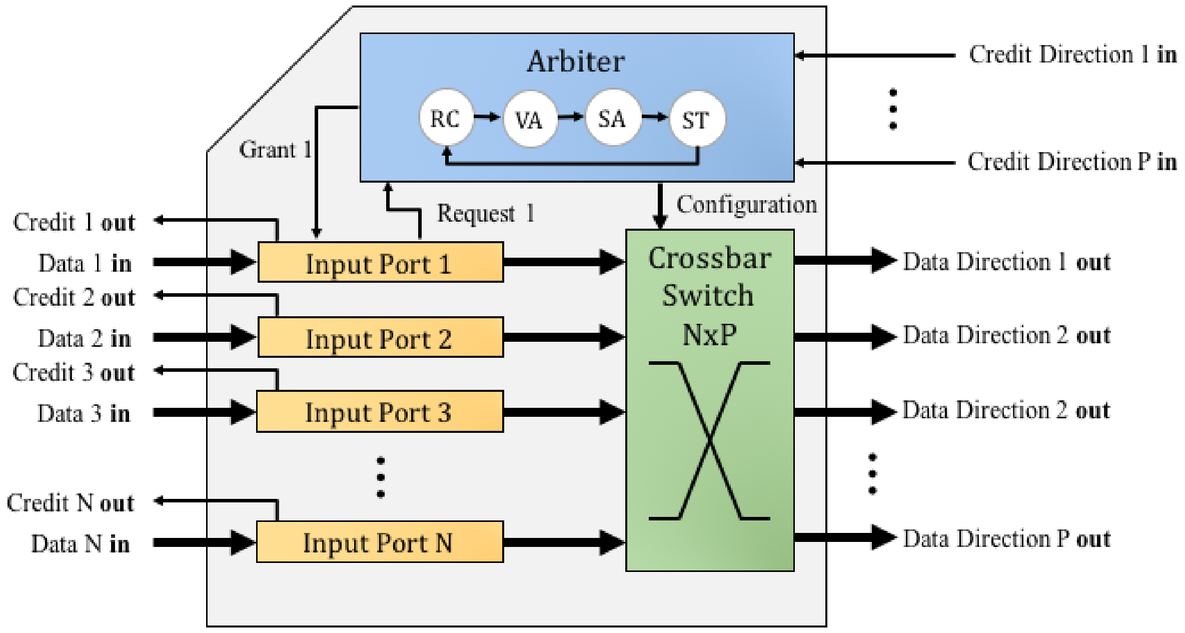


Figure 2.11 NoC router micro-architecture.

2.5.1 Crossbar

Crossbar switch is an asynchronous combinatorial logic sub-component of the router which has multiple input buses and output buses. It allows any input bus to be connected to at most one output bus. This enables data to traverse through the crossbar to any output bus. To control which input bus is connected to which output bus, an input configuration bus is used. Crossbar is made up of many de-multiplexers (demux), where each demux is connected at the input to one of the input port of the crossbar, at the output to all router outputs, and the mux selection port is connected to configuration port from arbiter. If multi-cast crossbars are used, data from 1 input port can stream to multiple output ports. However, tri-state buffers need to be added to solve fan-out problem. In this work, we will only consider crossbars and NoCs without multi-cast feature.

2.5.2 Input port

This is the major component in the router since it consumes majority of the power and takes up majority of the chip area. Input port is essentially composed of a FIFO (no Virtual Channel) or multiple FIFOs and multiplexers (in case of >1 Virtual Channels). A D-Flip flop (primary 1-bit storage element) is made up multiple NAND and OR gates. While designing large storage elements, several flip-flops are arranged in an array and some fabrication fail components are added. Some of these components include flip-flop scanners to identify faulty flip-flops, and some design fuses to disconnect failed components permanently. Since 32 flip-flop are required to store 1 32-bit word, SRAM has large power consumption and takes up large chip area.

Since FIFOs is a type of buffer and are essentially made up of D-flip flops, it is crucial to manage their size and organization to save chip area and power. Number of flits a FIFO can store is known as its depth. The router can have one pool of memory for all input ports, or each input port can have one pool of memory for each virtual channel. In each case, the depth/share of each input port or virtual channel can be static or dynamic.

2.5.3 Arbiter

Arbiter stores and decides the configuration to set for the crossbar. It manages flit requests coming from the input ports and grants them as per the arbitration scheme. Arbiter contains the main brains and logic for entire router. Arbitration scheme can be static (decided during design) or dynamic (decided as per congestion/traffic during run-time). It makes the output port status ‘free’ as soon as the tail flit traverses and makes its status ‘occupied’ before the header flit traverses through the crossbar. In this work, we will consider several fixed priority arbiters and use their synthesis libraries.

2.6 NoC Mapping

While designing any embedded system with top-down approach, the application is first broken down into tasks and their data dependencies. This is represented in the form of Task Graph (TG). Task graph is a Directed Acyclic Graph (DAG) in which nodes represent tasks and edges represent the data dependency and execution sequence. Tasks are grouped and assigned to hardware execution blocks such as CPU, DMA, GPU, co-processor, Accelerator etc. This is called Task assignment or task mapping shown as second optimization stage in Figure 2.12. This problem falls under Hardware-Software Co-design domain where the designer is still deciding on which task to execute on hardware and which task to execute using software (ie. CPU/co-processor). In this work, we are not addressing task mapping problem. We will be addressing core mapping and configuration problem shown as 3rd and 4th optimization stage in Figure 2.12 NoC design workflow and this work’s targeted workflow.

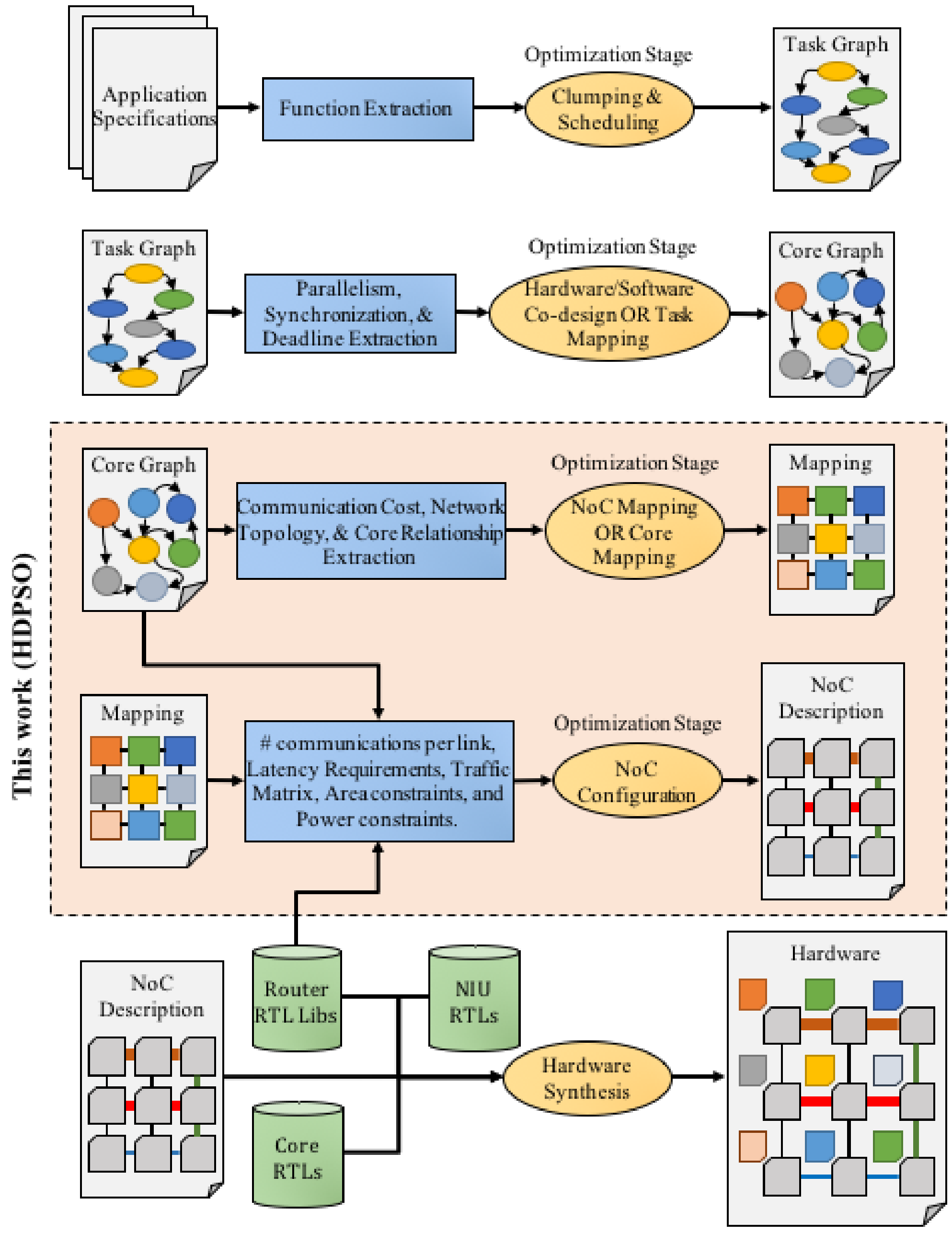

Application Specifications
Function Extraction
Optimization Stage
Clumping & Scheduling
Task Graph
Task Graph
Parallelism, Synchronization, & Deadline Extraction
Optimization Stage
Hardware/Software Co-design OR Task Mapping
Core Graph
This work (HDPSO)
Core Graph
Communication Cost, Network Topology, & Core Relationship Extraction
Optimization Stage
NoC Mapping OR Core Mapping
Mapping
Mapping
# communications per link, Latency Requirements, Traffic Matrix, Area constraints, and Power constraints.
Optimization Stage
NoC Configuration
NoC Description
NoC Description
Router RTL Libs
NIU RTLs
Core RTLs
Hardware Synthesis
Hardware

Figure 2.12 NoC design workflow and this work's targeted workflow.

After tasks are mapped onto hardware execution blocks/SIPs, the result is an Application Core Graph (ACG). An ACG is a DAG which contains hardware execution blocks/SIPs/Cores as nodes and their communication information in the form of edges connecting from one node to another. Assigning SIPs to locations (tiles) in a network topology is known as core mapping. Henceforth, we will be referring to core mapping problem as mapping problem. We define the NoC mapping as conversion of Application Core-

Graph (ACG) to a 2D-mesh NoC $m(x, y)$. The NoC mapping CAD tool determines a mapping $m(x, y)$ such that the overall NoC communication cost and energy are minimal. A mapping tool is efficient if it produces a minimum communication cost and energy NoC mapping solution with minimum number of iterations consuming least amount of compute time.

NoC mapping solutions are combinations/arrangements of cores in an order which corresponds to the tiles of a topology. Mapping IP blocks or cores onto a NoC topology becomes complex (Figure 2.13), as the size of NoC (search space) grows factorially with increase in the number of cores and topology tiles. This is shown for a small application which contains 4 cores to map which results in $4! = 4 \times 3 \times 2 \times 1 = 24$ possible mapping solutions out of which only one is optimal. The formula for size of the search space is given below:

$$Size\ of\ Search\ Space = n! \tag{1}$$

where n is total number of cores in an application

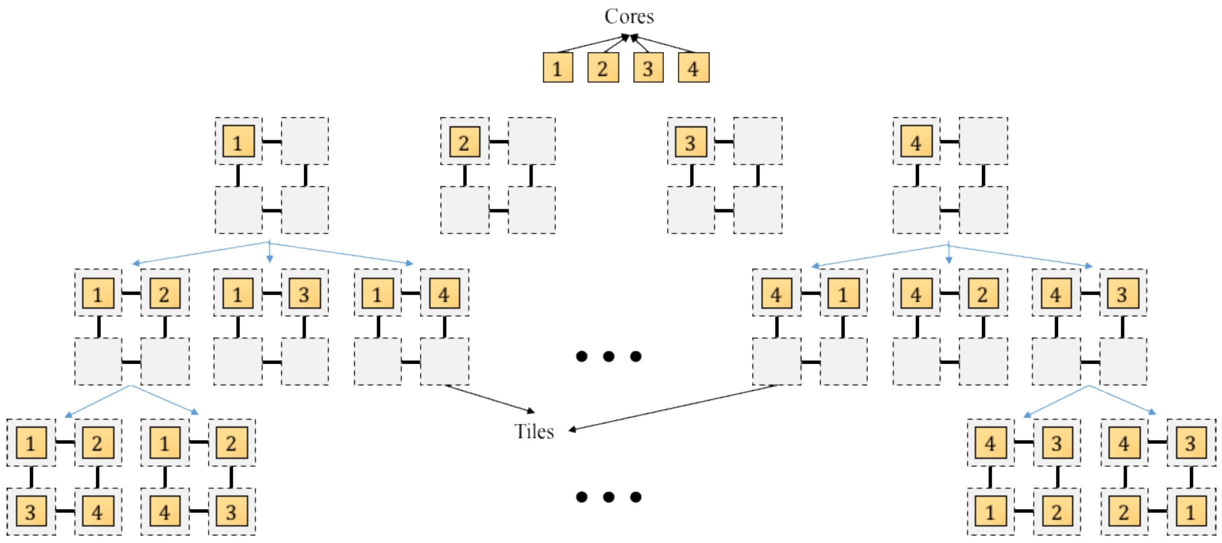


Figure 2.13 Mapping possibilities for a 2 by 2 mesh NoC with 4 cores.

Application mapping for 2D-mesh NoC is often considered as a constrained quadratic assignment problem of NP-hard complexity [19, 20, 21]. This means that the problem is not solvable in real time but it is verifiable in polynomial time. In other words, given a solution, we can check wether it is a good one or not but we cannot come up with a optimal solution within polynomial time. For example, we cannot directly come up with a best core mapping for a large application in short time. However, if someone provides a core mapping to check, we can easily check its cost. These kind of problems are best solved using stochastic algorithms which try and check many promising solutions before reaching a final solution.

Mapping is one of the major steps in NoC design where the cores are positioned onto topology nodes. It is crucial to choose an optimal mapping that results in minimal latency and energy consumption while maximizing the NoC bandwidth. A black box analogy of NoC mapping tool is shown in Figure 2.14.

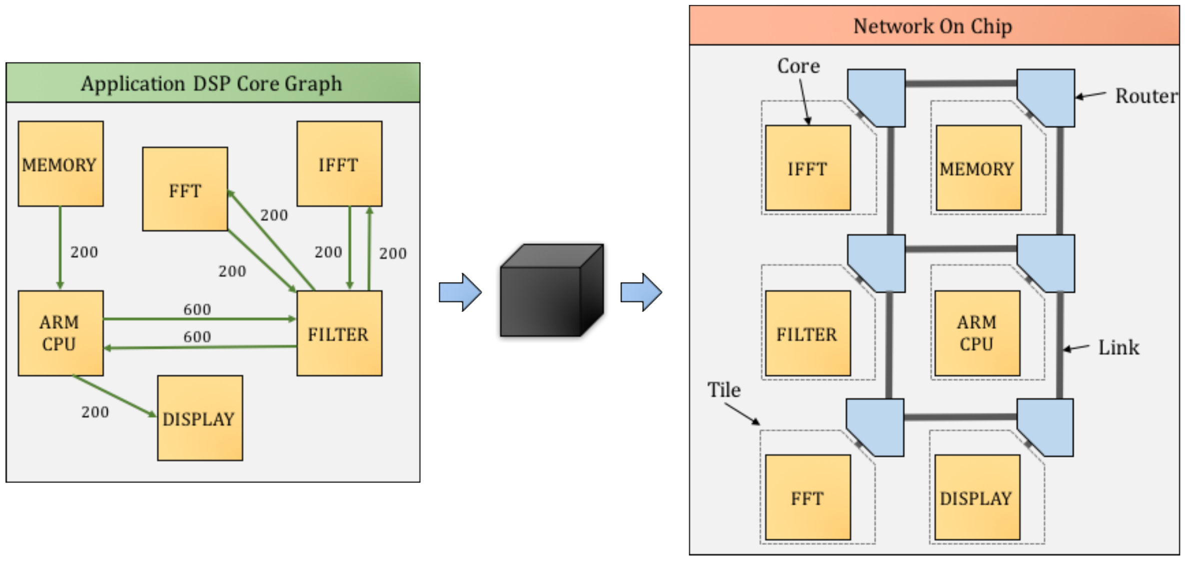


Figure 2.14 Black box analogy of NoC mapping tool.

2.6.1 State-of-the-Art Mapping tools

2.6.1.1 Deterministic Techniques

Jiang *et al.* have proposed to use a tree based branch-and-bound search technique to find all the combinations of possible solutions for a hybrid switched network [22]. During the execution of algorithm, each link in the hybrid network is permutated either to be circuit or packet switched. Finally, the best switching technique for each link is found. For deterministic algorithms, time to reach an optimal solution increases factorially with the solution space. Their methodology also consumes more time as compared to NMAP [23], NormalBB [21], and DPSO [24] techniques. Another popular deterministic technique is Mixed Integer Linear Program (MILP), which depends on iterations of linear equations of integers to solve the optimization problem. Bender has proposed an MILP model which determines optimal mapping based on execution time, processor cost, and communication cost [25].

2.6.1.2 Stochastic Techniques

Among the multi-dimensional or large search space evolutionary algorithms, the Genetic Algorithm (GA) is an interesting alternative to produce near-optimal solutions. Morgan *et al.* have used GA to optimize NoC mapping with a complex cost function involving power, chip-area, latency, and reliability [20]. GA based optimization techniques are compared extensively with other popular optimization techniques by Sahu and Chattopadhyay [26] and found that GA algorithm is not suitable for NoC mapping solution space since it converges using a single point rather than two or three reference points as in particle swarm (ie. local-best and global-best).

Yang *et al.* have presented an accelerated simulated annealing (SA) technique, which initially executes a few iterations in branch-and-bound method to reach a temperature called t_k and then iterates through the temperatures using traditional SA algorithm [27]. As opposed to other mapping algorithms, their t_k-SA technique focuses on generating better initial heuristics. However, their algorithm has been applied only to homogeneous NoC cores.

In Tabu Search (TS) techniques, a short-term memory is used to remember the previously visited solution space and put a "Tabu" on them, meaning the algorithm is forbidden to visit that space again. Tino and Khan have employed Tabu search to map application cores to an application specific NoC topology [28]. They intend to optimize power, and performance where the NoC performance is evaluated based on the dynamic factors such as contention. Each time a new solution is generated, it is analyzed by invoking the floor-planner that minimizes the wire length and area along with providing the power consumption. The newly proposed solution is accepted and memorized if it is present in the neighborhood or in the Aspiration list but not in the Tabu list. The algorithm stops as soon as it finds a solution that satisfies the stopping criteria. However, there may be other solutions, which result in lower power and area consumption.

Nedjah *et al.* have proposed a structured representation of the task graph and IP library [29, 30].We use similar XML representations as input to our tool. Task assignment problem, in which tasks are mapped onto IPs, precedes mapping problem. Previous mapping tools first come up with an optimum assignment and then find optimal mapping for it. On the other hand, their technique uses several assignment solutions instead of one to generate several mappings, which allows them to trade-off the design objectives of interest. The cost function is composed of hardware area, execution time (deadline criteria), and power consumption.

In assignment and mapping optimization, they use well-known GA algorithm type NSGA-II, CAFES, and micro-GA to compare the results. microGA worked better in saving the chip area and power up to 80% but at the cost of higher application (tasks) execution time of up to 600%. However, it can explore more search space in less amount of time as compared to NSGA-II.

Particle swarm optimization is a population-based technique inspired by social behavior of a group of animals that was developed in 1995 by Eberhart and Kennedy [8]. Each particle is a candidate solution and new solutions are generated based on the particle's own best-found optimal solution (local best) and swarm's best optimal solution (global best) [24]. Transformation from an old solution to a new solution is known as particle movement. It has more exploitation capabilities and more stable as compared to Genetic Algorithm (GA). This is because in GA, new offspring (current solution) depends just on its parent(s) (previous solution), but in DPSO a new solution depends on its local best and global best. In the beginning of DPSO algorithm, particles spawn in different promising search space regions. As the algorithm iterates or progresses through time, the particle moves in the search space looking for optimal solution. It also remembers its previous best fit positions, while learning from the past experiences of other particles. The quality of a particle is evaluated based on its fitness such as cost. We are proposing an extended formulation for our HDPSO based NoC mapping methodology.

As with all the stochastic optimization techniques, the DPSO might not find the optimal solution as it does not search the entire search space. Probability of catching an optimal solution mainly depends on the initial heuristics that place the search agents (or particles) in the search space. A good heuristic algorithm might place particles in some promising regions of the search space and find an optimal solution early. In most of the applications, DPSO particles spend a large amount of time exploring the regions of space already explored by themselves or by the other particles. This causes the algorithm to waste time and iterations in determining the cost of the same solution again and again.

In the past, there have been many modifications done to the baseline DPSO algorithm to improve or cater to a specific application. On such attempt was done by Li *et al.* [31] where they have proposed an improved particle swarm optimization algorithm which embeds a repulsion operator and local searching operator for each particle to induce swarm diversity and local refinement. This repulsion operator is enhanced and used in our tabu-ed reflection mechanism described in Section 3.3.4. They have used an XY routing mechanism and optimize the NoC for either communication power consumption or communication delay due to link load. However, their algorithm does not perform multi-objective optimization resulting in a solution with optimal communication power and delay. We have adopted a similar formulation for our HDPSO based NoC mapping methodology.

2.7 NoC Configuration

In addition to NoC mapping, next major steps in designing NoC include finding paths for the traffic flows while reserving resources across the NoC and determining NoC architectural parameters, such as the data width of the links, buffer sizes used in the router, and frequency of operation [32]. After cores are mapped to their respective nodes in topology, each communication edge in the core graph has to be assigned a physical path (links) with a bandwidth high-enough to support it. Then the resources along these paths are reserved to support edges communicating through them. Resources should be allocated generously to be able to support the collective bandwidth required by the edges communicating along the path, but at the same time to have minimum chip area and power consumption. These conflicting requirements give rise to the configuration optimization of NoC. Since there are many different possibilities (combinations) of architectural parameters that can satisfy a given application bandwidth on a given topology, there is a need to find an optimal solution. On the same lines of NoC mapping, NoC configuration solution space also grows factorially with an increase in the number of cores and library component options available. Therefore, there is a need to employ a stochastic technique to consider promising solutions rather than developing a deterministic methodology.

2.7.1 State-of-the-art NoC Configuration Tools and Work-flows

NoC mapping and configuration are two separate but linked optimization problems, which have an input-output relationship. NoC mapping optimization produces input for the NoC configuration optimization. NoC mapping optimization problem has been exhaustively researched and various optimization techniques have been employed [19, 33, 20, 21, 24, 32].

Æthereal design flow for NoC synthesis was proposed by Philips/NXP research along with Goossens et al. [34]. It offers operational design flow to dimension and generate application specific NoC instances and configurations [35]. Æthereal design flow, as shown in Figure 2.15, incorporates two optimization loops, one for NoC mapping and another for NoC configuration. In this design flow, only buffer sizes are optimized. Stergiou *et al.* have proposed another NoC synthesis oriented design library called Xpipes Lite which aids in automatic generation of heterogeneous NoCs [36].

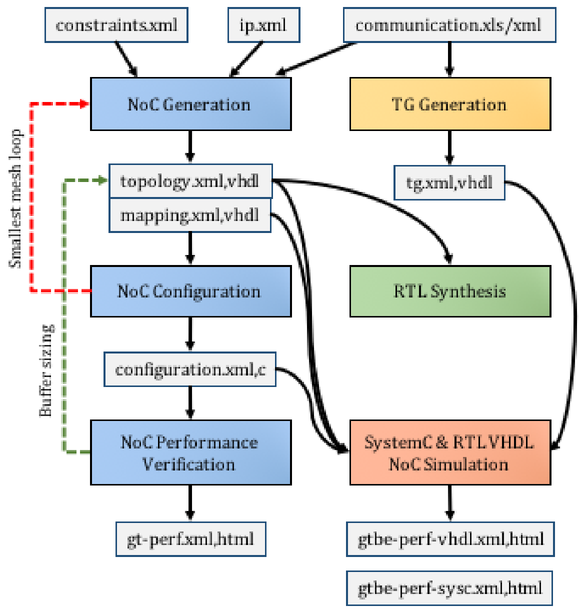


Figure 2.15 Æthereal NoC configuration and mapping workflow.

NoCTweak [37] is a highly parameterized NoC simulator which can simulate some embedded or synthetic traffics including (uniform random, tornado, shuffle, hotspot, etc.) for large network sizes. It combines usage of cycle-accurate SystemC models of routers along with router RTL coded in VHDL and synthesized using 65nm CMOS standard cell library. The simulator reports packet delay, network throughput, average power of router and each of its components, global clock tree and PLL, and energy consumption per packet. However, it does not allow NoC designers to expand the RTL library and provide their own custom router sub-components. The tool can only simulate for 65 nm technology as it was coded using models of 65 nm components.

Another major drawback is that while it does provide energy and power information of the NoC, it does not allocate or configure the routers so that the network occupies minimum chip area and consumes minimum power. Buffer sizes in the tool are modifiable manually when starting the simulator but it modifies the buffer depth for all routers in the network. This homogenous increase in buffer depth help just uniform random traffic but critical power and area savings opportunities are missed if the application does not have a traffic pattern of uniform random. NoCTweak allows custom applications to be imported and simulated, however it only performs mapping of the application using NMAP [23] optimization technique. NMAP is a good stochastic technique but further researches have outperformed it on many fronts [26].

Intelligent buffer merging technique is presented by Tsai *et al.* [38] where each routers input port load is calculated by using a traffic matrix and mapping the coregraph edges onto physical links. This edge-to-link mapping gives rise to a bandwidth constraint which should be satisfied by the input port configuration algorithm. Since they consider adaptive routing mechanism, data from a single source core can arrive from two input ports (two routes) and put into the same buffer. The edge-to-link mapping algorithm is improved and used in our configuration HDPSO algorithm described in section 4.8.1.

Chapter 3

Optimal NoC Mapping using HDPSO

3.1 Introduction

PSO [8] is an optimization method that optimizes a problem by first coming up with random solutions and then iteratively improving them by using basic equation of motions (ie. distance, velocity, and acceleration of particles). A candidate solution is called a particle and the all-time best candidate solution is known as global-best particle. Collection of particles which are linked together by common global best-particle is called a swarm.

DPSO is used for optimization of a discrete search space. A discrete search space implies that the solution is a combination, a permutation or a quantized quantity. For NoC mapping, the search space represents all the combinations of cores mapped to different tiles. Therefore, DPSO can be used to find the optimal mapping of cores to a suitable size 2D-mesh NoC. Traditionally, DPSO for NoC mapping starts with an initial population (particle) generated through some heuristic method to predict promising regions of the search space. Then the iterations cause the particle to evaluate its current cost and swarm the best global cost. The operation of a typical DPSO mechanism is shown in Figure 3.1.

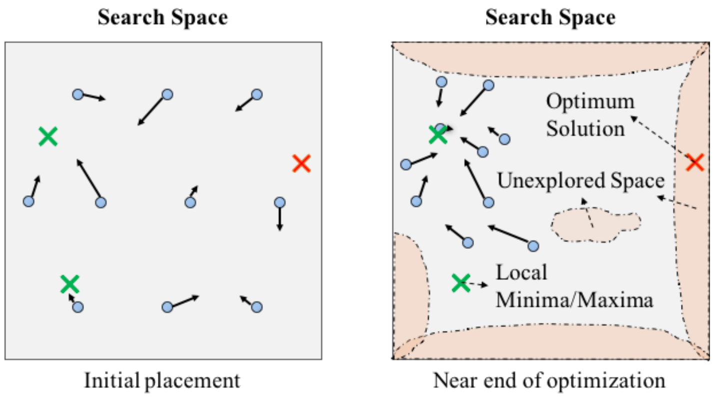


Figure 3.1 DPSO Search spaces – particles reach sub-optimal solutions

A major issue with the DPSO based methodology is the particles that waste the computing resources and time while staying in the previously explored solution space rather than exploring the un-chartered solution space. We propose a hybrid discrete particle swarm optimization methodology that overcomes this shortcoming. In our mechanism, we add the concept of Tabu-search to discourage the particles (or search agents) to pass through the solution space that has been already explored. The overall detail of our methodology having four stages is presented in Figure 3.2.

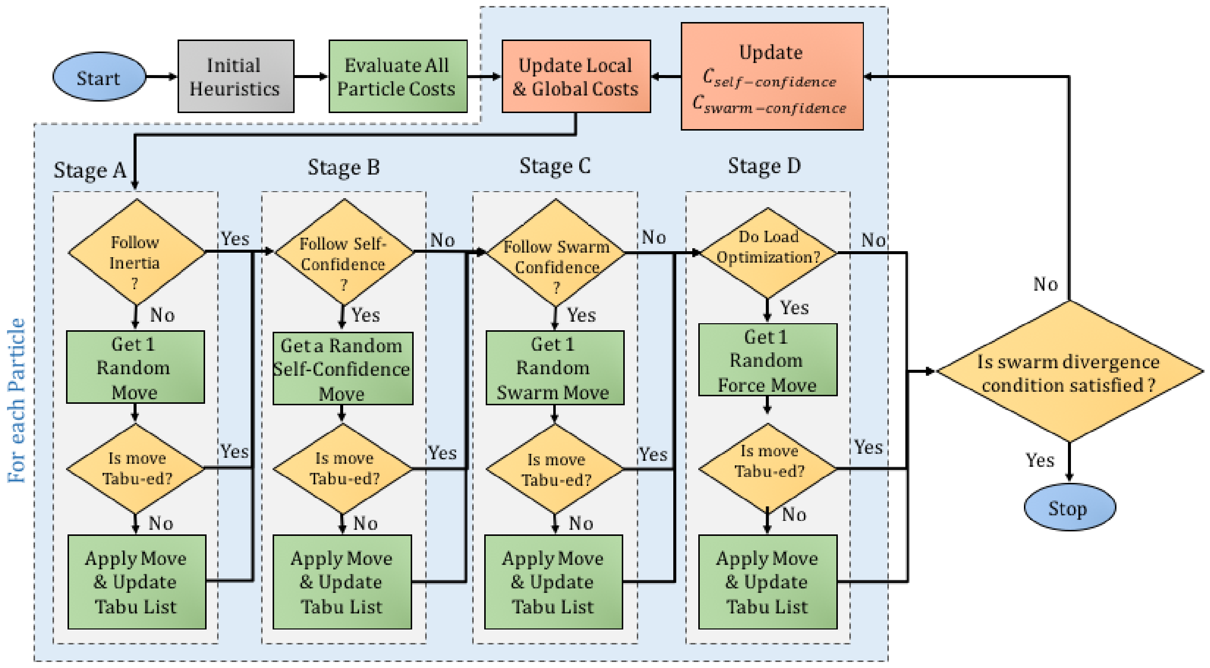

Start
Initial Heuristics
Evaluate All Particle Costs
Update Local & Global Costs
Update $C_{self-confidence}$ $C_{swarm-confidence}$
Stage A
Stage B
Stage C
Stage D
For each Particle
Follow Inertia ?
Yes
No
Get 1 Random Move
Is move Tabu-ed?
Yes
No
Apply Move & Update Tabu List
Follow Self-Confidence ?
No
Yes
Get a Random Self-Confidence Move
Is move Tabu-ed?
Yes
No
Apply Move & Update Tabu List
Follow Swarm Confidence ?
No
Yes
Get 1 Random Swarm Move
Is move Tabu-ed?
Yes
No
Apply Move & Update Tabu List
Do Load Optimization?
No
Yes
Get 1 Random Force Move
Is move Tabu-ed?
Yes
No
Apply Move & Update Tabu List
No
Is swarm divergence condition satisfied ?
Yes
Stop

Figure 3.2 HDPSO Algorithm.

In each iteration, a particle's new position is calculated based on three of particle's own properties and one global swarm property. Consider the position of the particle at iteration k i.e. $p_k = <p_{k1}, p_{k2}, p_{k3} \cdots p_{kn}>$ where p_{kn} represents a tile and n indicates the tile number. The iterative formulation for the HDPSO particle is given in the following equation:

$$p_{k+1} = \begin{pmatrix} C_{inertia} \times I \\ \oplus \\ C_{self-confidence} \times (p_k \longrightarrow local_{best}) \\ \oplus \\ C_{swarm-confidence} \times (p_k \longrightarrow global_{best}) \\ \oplus \\ C_{load-opt} \times (p_k \longrightarrow load_{optimized}) \end{pmatrix} \times p_k \qquad (2)$$

where p_{k+1} and p_k represent next generation and current particle core arrangements respectively

$a \rightarrow b$ is a sequence of swaps applied on components of a to transform it to b (see Figure 3.5)

$\oplus$ is a fusion operator. For two sequences a and b. $a \oplus b$ is the sequence in which the sequence of swap of a is followed by sequence of swaps of b

$C_{inertia}$ is inertia constant that determines the particle's willingness to stay at the current solution or move on to another solution

$C_{self-confidence}$ is self-confidence that determines the particle's willingness to move closer to the local best

$C_{swarm-confidence}$ is swarm confidence that controls the particle's willingness to move to the global best solution

$C_{load-opt}$ is the communication volume optimization coefficient that controls the optimization of particle's communication volume list

I are the Identity swaps such as *swap(1,1), swap(2,2)…… swap(n,n).*

$local_{best}$ is the previous local best core arrangement of the particle

$global_{best}$ is the swarm best core arrangement

$load_{optimized}$ represents the core arrangement after applying communication volume optimization on p_k

In each iteration, the particle might take a step towards a random new location, local best, global best, communication volume optimum mapping or a combination of these. A particle falls through a ladder of decision points where a random number is generated and it is decided whether the particle should take the path or not. The decision is biased by inertia constant $C_{inertia}$, self-confidence $C_{self-confidence}$, swarm confidence $C_{swarm-confidence}$, and communication volume optimization coefficient value $C_{load-opt}$. The details of the main steps of our HDPSO based NoC mapping methodology are presented in this section. The stages are basically core swaps depending on various parameters such as random inertia, self-confidence, swarm confidence and communication volume optimization. These are described later in the Evolution of Swarm Particle (solution) in Section 3.3.6.

3.2 Problem Definition

We define the NoC mapping of an Application CoreGraph (ACG), $ACG(AM)$ onto a 2D-mesh NoC, $m(x,y)$. The NoC mapping CAD tool determines a mapping $m(x,y)$ such that the overall NoC communication cost are minimal. A mapping tool is efficient if it produces a minimum communication cost NoC mapping solution with minimum number of iterations consuming least amount of compute time. To simplify the mapping problem, X-Y routing technique is assumed along with round-robin arbitration for determining the cost parameters. We assume a regular mesh topology where the cost for the mapping solution or particle (P_{Cost}) can be defined as:

$$P_{Cost} = \sum_{i=0}^{size(AM)} \sum_{j=0}^{size(e)} Vol(e_i) \times HopCount(e_i) \qquad (3)$$

where e is the collection of edges in a core graph,
$Vol(e_i)$ is the communication volume (MB/s) on edge i,
$HopCount(e_i)$ is the hop distance between two communicating cores (or vertices).
$size(AM)$ indicates the number of application modes

An application can have many modes of operation and in each mode of operation, SoC has different communication characteristics among cores. A low-power mode often reduces the communication volume among cores and that can be represented by a separate coregraph. In other words, a single application can have many coregraphs. Additionally, in some application modes, not all cores might be active or operational. The inactive cores might be power-collapsed in order to save power and reduce battery consumption. These power-collapsed cores are no longer in the coregraph (because they do not consume or produce network traffic) but the router is still operational to allow data transit. An example of one such multi-modal application is shown in Figure 3.3. AM 1,2,3, and 4 are low/high performance modes, while in AM 5, core 2 is power collapsed, and in AM 6 cores 3 and 6 are power collapsed.

For a given mapping, we apply cost formula to each mode (coregraph) and sum the results together to get total cost. The resulting sum cost gives equal weight to all application modes. Since not all application modes are operational at all times, NoC designer might decide to optimize NoC cost more for one application mode than other. This can be taken care of by providing the cost_weight parameter along with each application mode (coregraph). This parameter will be used to normalize the cost of each application mode before summing all costs.

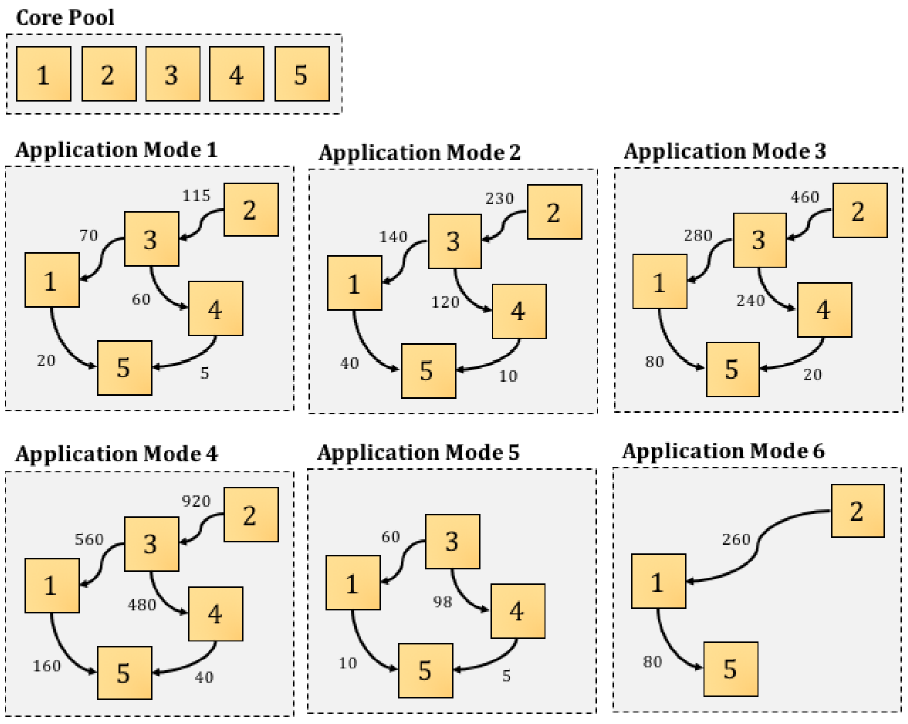


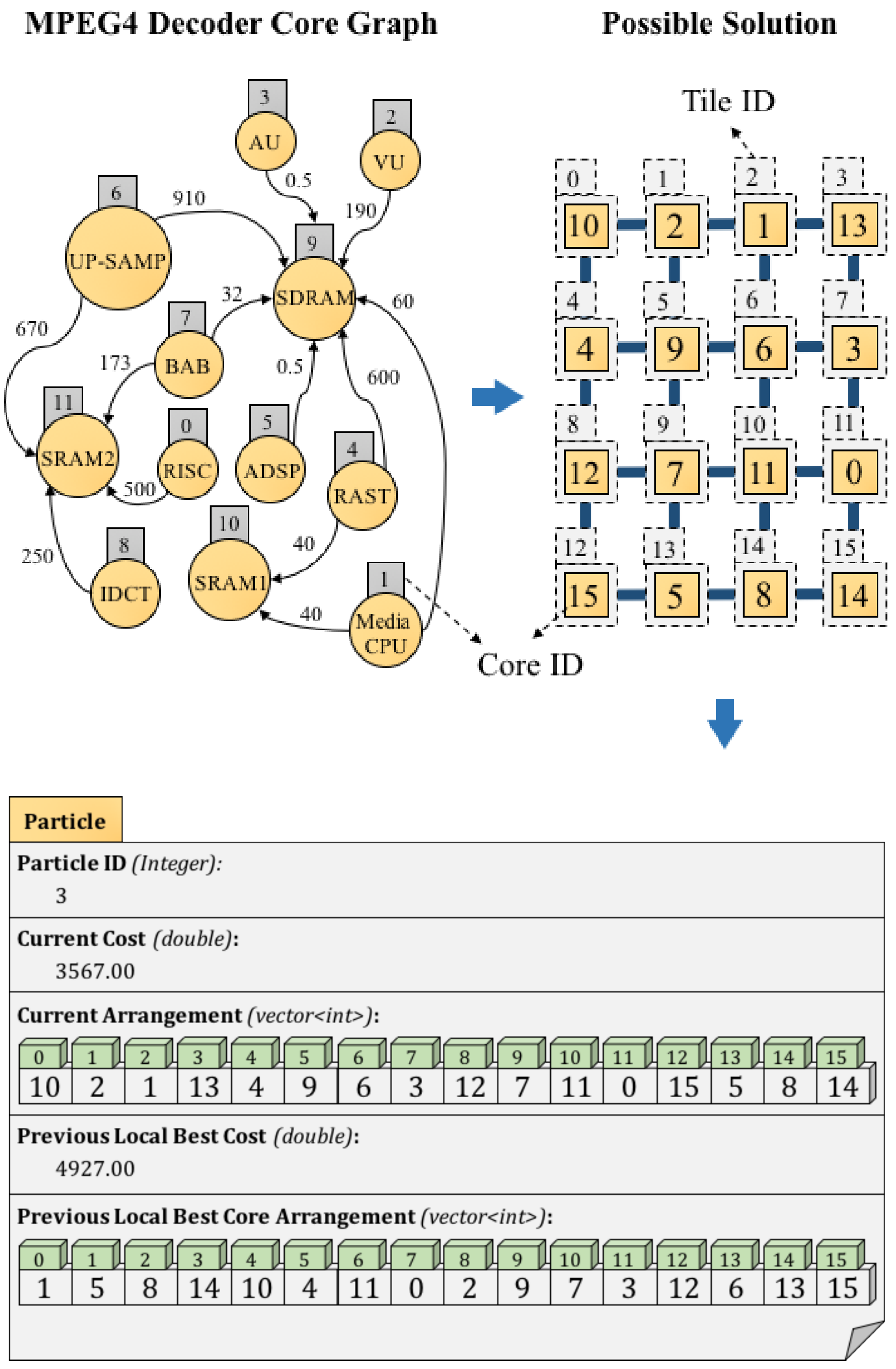
Figure 3.3 Application with multiple modes of operation (AM).

3.3 HDPSO Algorithm

3.3.1 A HDPSO Particle

An HDPSO particle is a candidate solution which contains possible arrangement of cores mapped to a 2D-mesh NoC. The overall cost is evaluated and calculated for each particle to determine whether the particle represents a better solution than the current global best solution. A particle stores its current cost, previous local best cost and core arrangement as well as the particle ID. As the particle moves and if it encounters a core arrangement (mapping) which has lower cost than its previous local best cost, it overwrites the previous best cost with the newly found cost and stores this new core arrangement as the previous local best core arrangement. This is incorporated in “Update Local Costs” function in Figure 3.2. The size of the particle determines the number of tiles available in the 2D-mesh NoC. Core arrangement is a vector representing the mapping of each core to a tile position in the NoC mesh. The structure of the particle is versatile and any NoC topology can be mapped by employing the same particle structure. Figure 3.4 shows a hybrid DPSO particle structure and an arbitrary mapping for MPEG4 core graph mapped to a 4x4 mesh NoC. The coregraph of MPEG4 contains just 12 cores. In order to map the coregraph to a square mesh NoC, 4 dummy cores (ie. core id. 12, 13, 14, and 15) are added. Their cost is neither calculated nor affects the optimization method. In fact, it was observed that these dummy cores were pushed to the corners of the mesh by optimization tool as it can be seen in the possible solution in Figure 3.4. This is because the corner tiles have minimum number of paths to other cores. In other word, corner cores have less neighbors.

The particle holds the current cost (ie. 3567.00 in Figure 3.4) of the core arrangement. After each iteration, current arrangement of cores is evaluated and the resulting communication cost is placed in current cost of the particle.



Particle

Particle ID *(Integer)*:
3

Current Cost *(double)*:
3567.00

Current Arrangement *(vector<int>)*:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	2	1	13	4	9	6	3	12	7	11	0	15	5	8	14

Previous Local Best Cost *(double)*:
4927.00

Previous Local Best Core Arrangement *(vector<int>)*:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	5	8	14	10	4	11	0	2	9	7	3	12	6	13	15

Figure 3.4 A HDPSO particle structure for MPEG-4 decoder.

3.3.2 Initial Population Generation

In our HDPSO based mapping, first an application core graph is imported for NoC mapping, where the number of swarm particles varies with the size of particle (or solution). Particles are spawned in each root branch with the first NoC tile assigned to the core with the same ID as the ID of the swarm particle. For example, particle 12 will have 12th core mapped to the first (0th) tile. It ensures that all the initial particles are spaced away from each other as well as in different solution space areas. Starting by generating completely random initial solutions could place them in the same area and in close-proximity. By using a fixed first tile for initial solution, we ensure that the initial solutions are at least separated by first combination. The other remaining cores are mapped randomly to random tiles. It also ensures that bigger application core graphs have more search agents or particles to search different search space areas. The pseudo code of the initial population generation method is given below.

As with all stochastic algorithms, different initial conditions produce different results and can sometimes help the algorithm reach optimal solution faster. However, intelligent initial population heuristics can consume time and are finding partial solution to the problem by using deterministic method. As application size grows, so does the time complexity of the initial placement deterministic algorithm which destroys the whole purpose of using a stochastic algorithm. To support both large and small solutions spaces, we have used a simple initial population placement algorithm given in Algorithm 1.

Algorithm 1: Initial Population Generation

Input: non-negative integer *no_of_particles*
Output: vector of particle objects vector<Particle> *population*

```
1. vector<Particle> population
// Iterate over all the particles
2. for i = 0; i < no_of_particles; i++ do
3.         vector<int> initial_condition(PARTICLE_SIZE, -99)
           // Map the first tile to Core i
4.         initial_condition[0] = i
           // Create particle object with initial condition
5.         Particle p(I, PARTICLE_SIZE, initial_condition)
            // Add particle object to the population
6.         population.push_back(p)
7. end
8. return population
```

3.3.3 Particle Movement (Applying Swap Moves)

In every iteration of a traditional DPSO method, a particle compares its current cost with its previous found best cost (local best) and swarm found best cost (global best) to determine the new position. However, in the NoC mapping problem, exact locations of particles in search space is not known. Distances between particles can be defined as number of swap moves required to make them equal as shown in Figure 3.5. In HDPSO, the new position of a particle is determined by comparing its current core arrangement p_k with the previous local best core arrangement l_{best}, global best core arrangement g_{best}, and communication volume optimization applied core arrangement f_{best}. This comparison results in a vector of swap moves which can be applied to particle A, for bringing it closer to the other particle B i.e. to. reduce the distance between particles A and B. However, all the swap moves are not applied directly in one iteration to avoid the particles to hop in the search space and might miss a potentially optimal solution. Instead, we choose a random swap move (*chosen swap move*) from the list of swap moves.

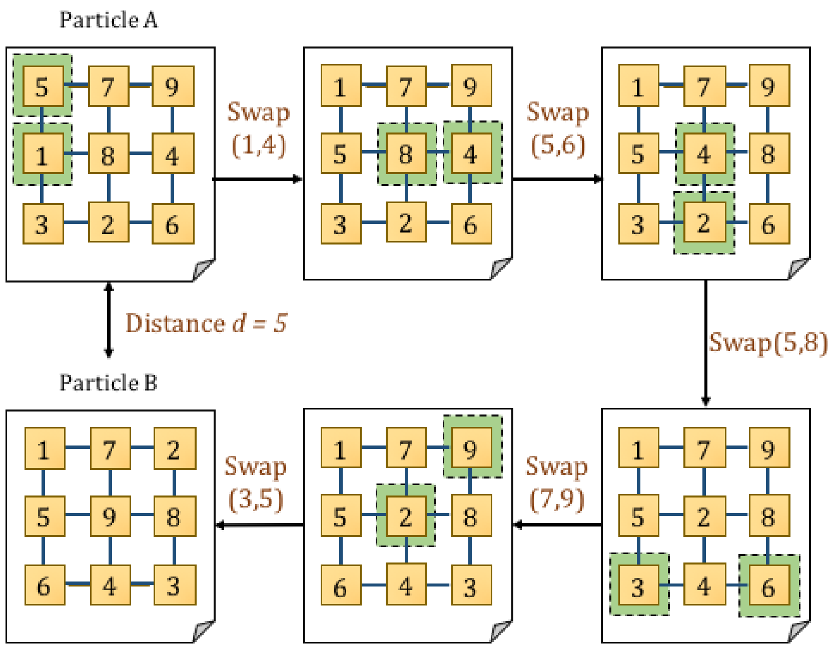


Figure 3.5 Determining distance between particles (particle A & B).

In the next step, the *chosen swap move* is applied onto a copy of the particle A and saved as temporary particle T. The swap move is not applied to the original particle A as it might result in a Tabu-ed particle (or solution). Tabu list is searched for any previous particle which has the same core arrangement as of particle T. If a match is found (the particle T is Tabu-ed), the *chosen swap move* is reflected and this step is repeated. If no match is found (particle T is not Tabu-ed) then the *chosen swap move* is applied to the original particle A. The whole process is illustrated in Figure 3.6.

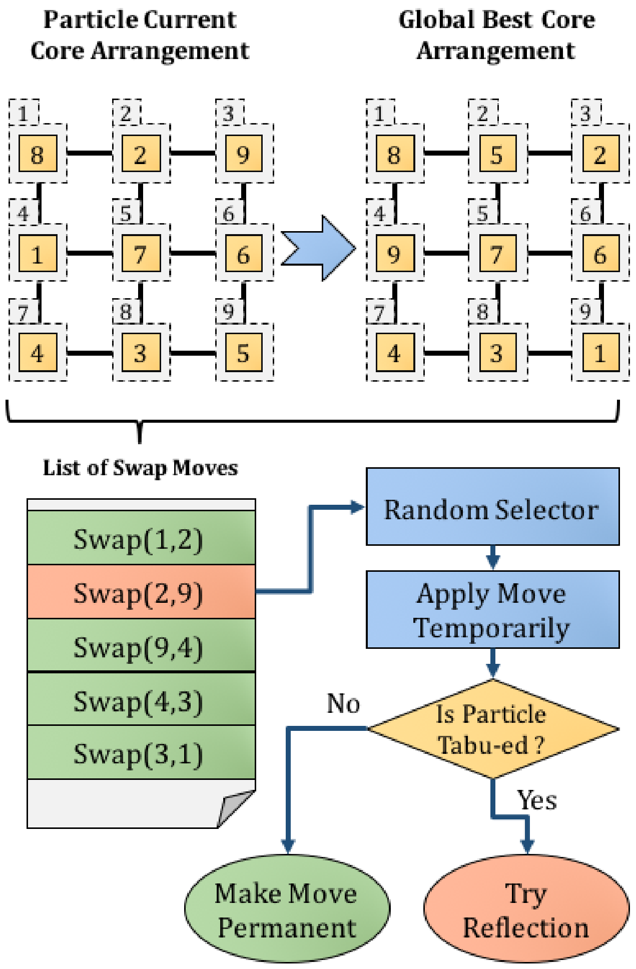


Figure 3.6 Identifying the chosen swap move.

In addition to the traditional swap move, we also propose more complex moves (see Figure 3.7) including transpose, horizontal shuffle, vertical shuffle, column swap, and row swap. A transpose move can only be applied to square mesh networks and it has an effect similar to that of matrix transpose. Horizontal and vertical shuffles nudge cores to n positions right or down respectively. Row swap and column swap moves exchange two columns or swaps. Each of these proposed moves can be represented in terms of smaller swap moves. In this work, we do not use these complex moves.

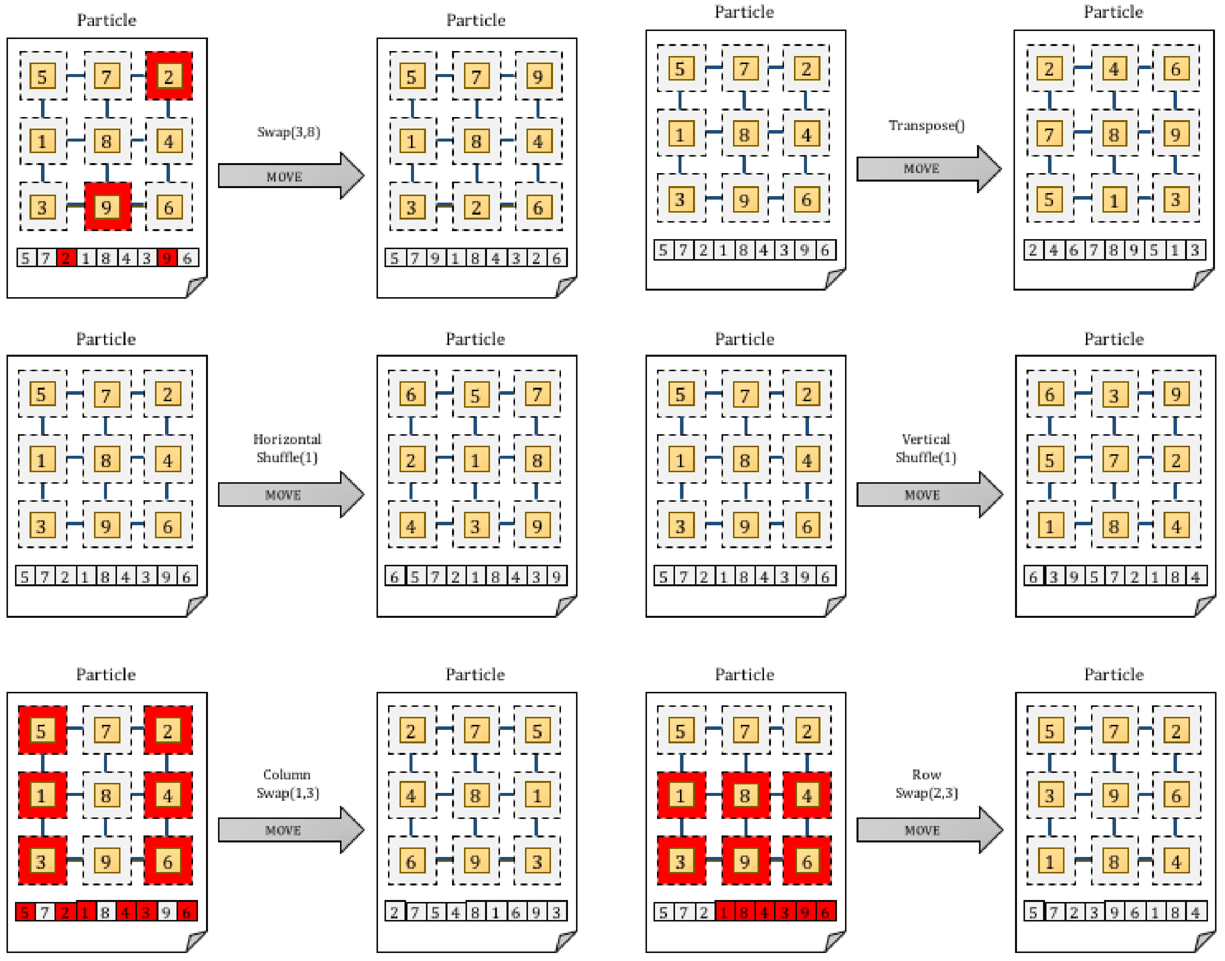


Figure 3.7 Complex swap moves.

3.3.4 Tabu-List and Particle Reflection

A short-term memory called Tabu-list is used to keep track of the previously visited search space areas and prevents the particles to enter the Tabu-ed zone. For large 2D-mesh NoCs having a large search space, the number of particles as well as iterations required to reach the optimal NoC mapping (solution) will grow. This will consume a large memory for the Tabu-list as more iterations are executed. Therefore, we keep the Tabu-list small and proportional to the number of particles as well as the number of iterations. In this way, the Tabu-list is operated as a short-term memory for the previously tried solutions. The size of tabu memory is not dependent on number of application modes. Size of the Tabu-list is determined by the following equation:

$$Tabu_{List_{size}} = \tau . N_{Particles} . N_{iterations} . N_{tiles} \qquad (4)$$

where $0 \leq \tau \leq 1$ is a tuning parameter chosen by the designer according to the computing system capabilities and its memory capacity. For this work, we consider $\tau = 1$

$N_{Particles}$ is the number of particles in search space

$N_{iterations}$ is the number of iterations to remember the particle position for

N_{tiles} is the number of tiles in each particle/mapping solution

Tabu literally means sacred or forbidden. In our proposed algorithm, solution points in the search space already visited by the particles are forbidden for them to visit again. We propose a reflection mechanism around the Tabu-ed search area as shown in Figure 3.8. The reflection mechanism ensures that all the particles that are not Tabu-ed will determine their moves. If the move is indeed Tabu-ed, then it proposes an alternative move. However, particles in swarm optimization should be allowed to enter the Tabu-ed area to support convergence towards the optimal solution. Therefore, a particle is allowed to apply the move on itself if all reflections are tried and all were either tabu-ed or invalid.

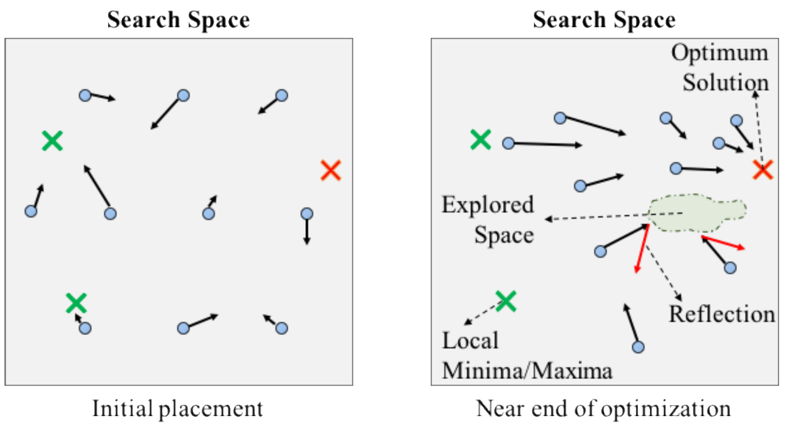


Figure 3.8 Reflection of particles from the explored search space.

After evaluating the new cost by the swarm particle, Tabu-list is appended with this new application mapping (core arrangement). It is quite common for the DPSO particles to show an oscillating behavior between two or more NoC mapping solutions. Swarm particles can get stuck in such situations and may remain in the barrier of similar costs for two different core mappings. A short-term Tabu memory is kept at the global level to remember the previously explored solution space or previously tried core mapping arrangements to prevent such oscillation behavior. When a proposed move results in a core mapping that has been already identified, the current iteration is not wasted. We intend to move such particles to explore the un-explored regions of the search space. The particles are moved by a random movement that can also be on the Tabu-list and huge amount of time would be spent on determining a random move that is not on the Tabu-list. Keeping particles in the Tabu-list for the past iterations make our HDPSO algorithm stable as particles do not try new core arrangements unless it is new. Variation of current cost in a DPSO particle is high as DPSO does not remember the previously tried solutions. On the other hand, HDPSO particle cost is stable due to the Tabu-list that keeps a list of previously tried solution as shown in Figure 3.9. These results indicate much lower level of oscillations. Lower oscillation reduce the number of computations performed in each iteration, thereby reducing the execution time.

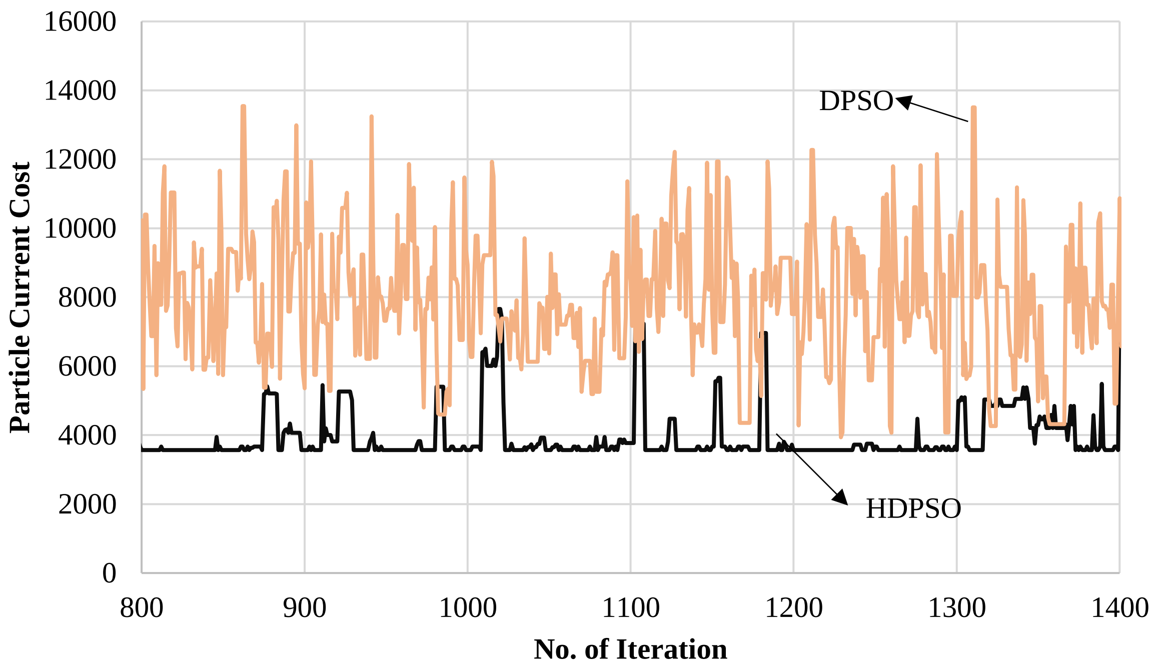


Figure 3.9 HDPSO vs DPSO particle cost variations for mapping of MPEG4 decoder.

We propose a particle reflection methodology to divert the particles towards un-explored search space areas instead of just random moves. Reflection refers to a change in the direction of the proposed move of a particle where one of the cores to swap remains the same but the other core changes. If a move is to result in an already explored mapping, a different move is proposed for the particle. Considering a 3x3

2D tiled NoC given in Figure 3.10. If the particle is trying to swap for tile 2 and 4 and the resulting mapping is Tabu-ed i.e. on the Tabu-list, then the move is reflected to perform the swap for NoC tile 4 and 8 as shown in Figure 3.10. The reflected-move is the reflection of the original move with respect to y-axis.

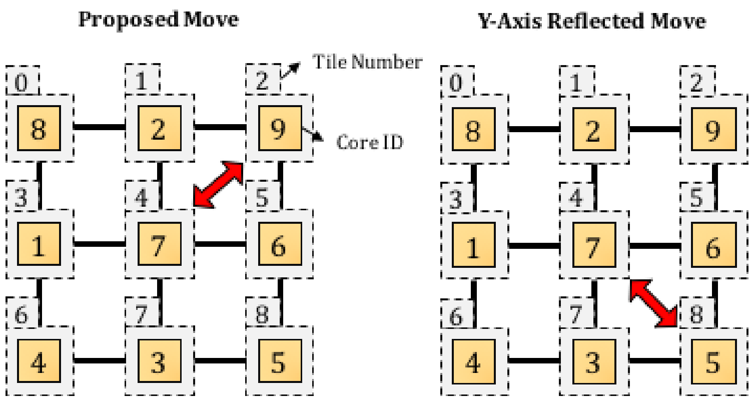


Figure 3.10 Reflection of a Tabu-ed swap (2,4) move.

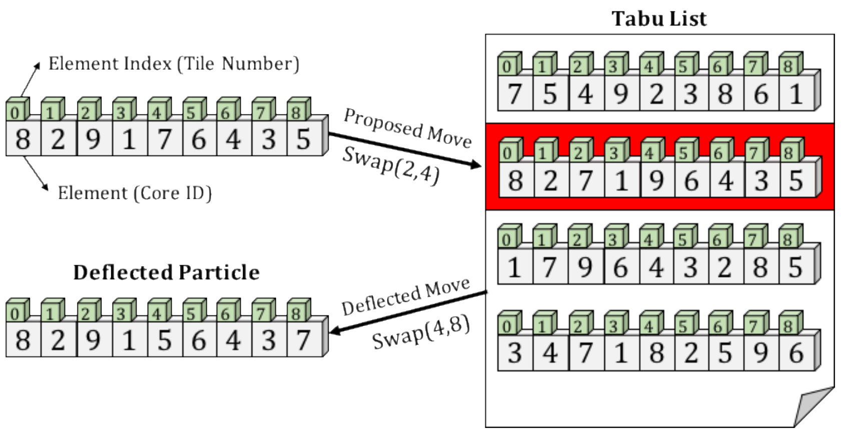


Figure 3.11 Reflection of a swap (2, 4) with respect to y-axis.

When a proposed move is predicted to result in a core arrangement already tried (i.e. Tabu-ed), the particle can be reflected in four ways. For a 3x3 mesh NoC some typical reflections are given below. $Move_{original}$ signifies the move which a particle makes to get closer to global best, local best, communication volume optimized solution, or just inertia. $Move_{reflected}$ means that the proposed original move has resulted in a core-arrangement (solution) which has already been tried and therefore another new untried move is proposed.

i. Reflection in x-axis: for $Move_{original} = swap(2, 4)$

$$Move_{reflected}(x - axis) = swap(4,0)$$

ii. Reflection in y-axis: for $Move_{original} = swap(2,4)$

$$Move_{reflected}(y - axis) = swap(4,8)$$

iii. Reflection in line $y = x$: for $Move_{original} = swap(1,4)$

$$Move_{reflected}(x - axis) = swap(4,3)$$

iv. Reflection in line $y = -x$: for $Move_{original} = swap(1,4)$

$$Move_{reflected}(x - axis) = swap(4,5)$$

3.3.5 Communication Volume based Optimization

We have devised an intelligent methodology to swap a near core that has smaller communication volume between itself and the source core with a far core, which has larger communication volume with source. After an application is imported for optimal NoC mapping, all the edges are arranged in ascending communication volume and placed in a vector list as shown in Figure 3.12 for a particular case. Each time a particle is evolved, hop distances for each of these edges is updated. The ascent in the hop distance means that the current edge, X has more communication volume than the previous edge Y but the cores of edge X are farther apart from each other than the Y edge. The working detail of the traffic optimization stage is shown in Figure 3.12 and listed below.

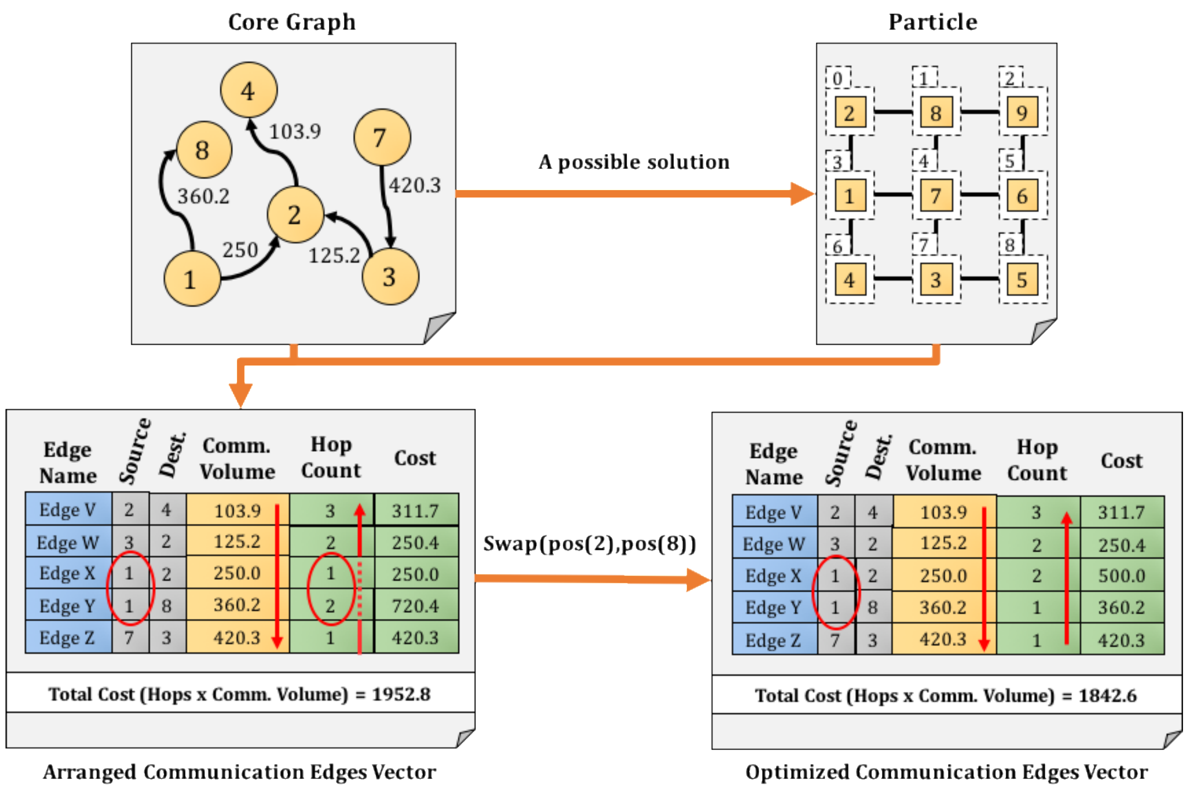
Step 1. Generate a list of arranged communication edges with the ascending communication volume.

Step 2. Find the edges with an ascending hop count that has the same source id. In Figure 3.12, X and Y edges ($1 \rightarrow 2$ and $1 \rightarrow 8$) have the same source core 1, and Y has higher communication volume and hop count than edge X.

Step 3. Swap the destination cores of these two edges. In Figure 3.12, core 8 should be closer to core 1 rather than core 2 because it has higher communication volume.

Step 4. Test if the swap move results in a Tabu-ed core arrangement. If it does, then try all the reflections. If the reflections are not possible or result is a Tabu-ed core arrangement, then apply the original move.

Step 5. Go to Step 2 until the end of edge list.



Edge Name	Source	Dest.	Comm. Volume	Hop Count	Cost
Edge V	2	4	103.9	3	311.7
Edge W	3	2	125.2	2	250.4
Edge X	1	2	250.0	1	250.0
Edge Y	1	8	360.2	2	720.4
Edge Z	7	3	420.3	1	420.3

Total Cost (Hops x Comm. Volume) = 1952.8

Edge Name	Source	Dest.	Comm. Volume	Hop Count	Cost
Edge V	2	4	103.9	3	311.7
Edge W	3	2	125.2	2	250.4
Edge X	1	2	250.0	2	500.0
Edge Y	1	8	360.2	1	360.2
Edge Z	7	3	420.3	1	420.3

Total Cost (Hops x Comm. Volume) = 1842.6

Figure 3.12 Communication Volume based core swapping.

3.3.6 Evolution of Particles

In the first iteration of our HDPSO methodology, the cost for all the particles is calculated and then overall local best and global best costs are determined. In later iterations of the optimization, a swarm particle passes through four stages as shown earlier in Figure 2. Before each stage begins, a particle decides to go through that stage or not using stage path decider. This stage path decider is based on biased random number generator. A stage path decider which is biased by self-confidence parameter $c_{self-confidence}$ is shown in Figure 3.13.

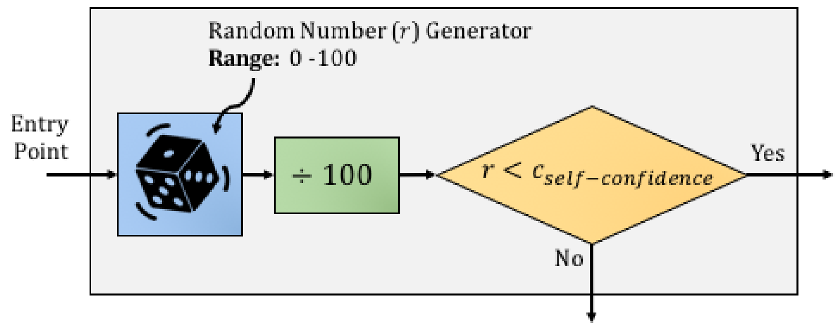


Figure 3.13 Local Convergence Stage (Stage B) path decider.

Stages that a particle may or may not go through, depending on path decider, are given below in sequence. Due to the nature of path decider, a particle may not go through any stage or may go through all of them. So in essence, a particle may not move at all or move to a maximum of 4 distance (4 swap moves) in one iteration.

3.3.6.1 Inertia Stage (Stage A)

In this stage, first it is decided whether to apply a random swap move or not. The move is based on the inertia probability constant, $C_{inertia}$. The identification of a random swap move is also illustrated in Figure 5.

1. If the decision is made to generate the random swap move then verify that the resulting core map is Tabu-ed or not.
2. Try reflections if the resulting core mapping belongs to the Tabu-list.
3. If the reflections are not possible or they result in a Tabu-ed core arrangement, then apply the original random swap move.

3.3.6.2 Local Convergence Stage (Stage B)

In this stage, it is decided whether to converge to a local best core mapping or not. The move is based on a self-confidence probability, $C_{self-confidence}$. If yes, then generate a list of swap moves by comparing the local best core mapping and the current core mapping represented by the swarm particle.

1. Select a random swap move to apply from the list generated. If the move results in a Tabu-ed core arrangement, then try reflections.

2. If all the reflections are not possible or result in a Tabu-ed core arrangement, then apply the original random move.

3.3.6.3 Swarm Convergence Stage (Stage C)

In this stage, it is determined whether the solution (swarm particle) converges to a global best core arrangement or not. The move is based on a swarm-confidence probability, $C_{swarm-confidence}$.

1. Generate a list of swap moves by comparing the global best core arrangement and current core arrangement of the particle.
2. Select one random swap move to apply from the list generated in the earlier step.
3. Test if the move results in a Tabu-ed core arrangement. If yes, then try reflections. If all reflections are not possible or result in a Tabu-ed core arrangement, apply the original random move.

3.3.6.4 Traffic Load Optimization Stage (Stage D)

In the final stage, it is decided whether to optimize the mapping by applying communication volume based core swaps or not. The decision is based on the load optimization probability, $C_{load-opt}$. A random number is generated between 0 and 100 and checked against load optimization probability. If the number generated is less than the probability, then communication volume based core swaps are applied. The algorithm of this stage is explained in Section 3.3.5.

Our HDPSO based NoC mapping process can be terminated after executing a fixed number of iterations or after a pre-determined number of swarm particles have converged to a common mapping.

3.3.7 Stopping Criteria

It is observed that the swarm particles of DPSO converge to an optimal point in search space near the end of optimization. It is at this point, the particles have settled and optimization need not continue. To detect this situation and stop the optimization loop, diversity factor proposed by Silva and Bastos-Filho [39] is modified and used. First, an average distance of each particle from other particles is calculated using the following equation.

$$d_i = \frac{1}{N-1} \sum_{j=1, j \neq i}^{N} dist(p_i, p_j) \tag{5}$$

where N is the number of particles.

$dist()$ returns the number of swap moves required to make particle i equals to particle j.

d_{min} and d_{max} are the maximum and minimum average distances respectively.

Since NoC mapping problem is a discrete combinatorial problem, Euclidean distance formula cannot be applied to get distance between two particles. Instead distance between two particles is given by the number of swap moves require to make both particles same as shown in Figure 3.5 Determining distance between particles (particle A & B).. Then diversity factor of each particle is calculated using the equation below:

$$DF_i = \frac{d_i - d_{min}}{d_{max} - d_{min}} \tag{6}$$

where DF_i is the diversity factor of particle i.

d_i is the average distance of particle i to other particles.

d_{min} and d_{max} are the maximum and minimum average distances respectively.

Optimization loop can be stopped when the following criteria given in the following equation is met:

$$\varpi \leq \frac{1}{N}\sum_{i=1}^{N} DF_i \tag{7}$$

where N is the number of particles.

ϖ is an empirically selected stopping diversity factor.

Choosing a random swap move from a list of moves results in slow progression towards the desired solution space region while exploring it. This is evident from the global cost comparison presented in Figure 3.14 for our HDPSO algorithm with the DPSO for a 3-mode 17-core application. In HDPSO, swarm particles take small steps towards the local or global best core arrangement rather than a sudden stride as shown in Figure 3.14. In the same number of iterations, HDPSO finds a better and lower cost arrangement faster than the traditional DPSO. Our HDPSO methodology executes about 50% faster and it has around 20% better cost as compared to DPSO. By allowing smaller swap movements, the HDPSO algorithm steps gradually towards the best cost rather than sudden jumps seen in DPSO. This also allows the HDPSO method to explore more parts of the search space, which enhances the chance of finding a better solution than the DPSO technique.

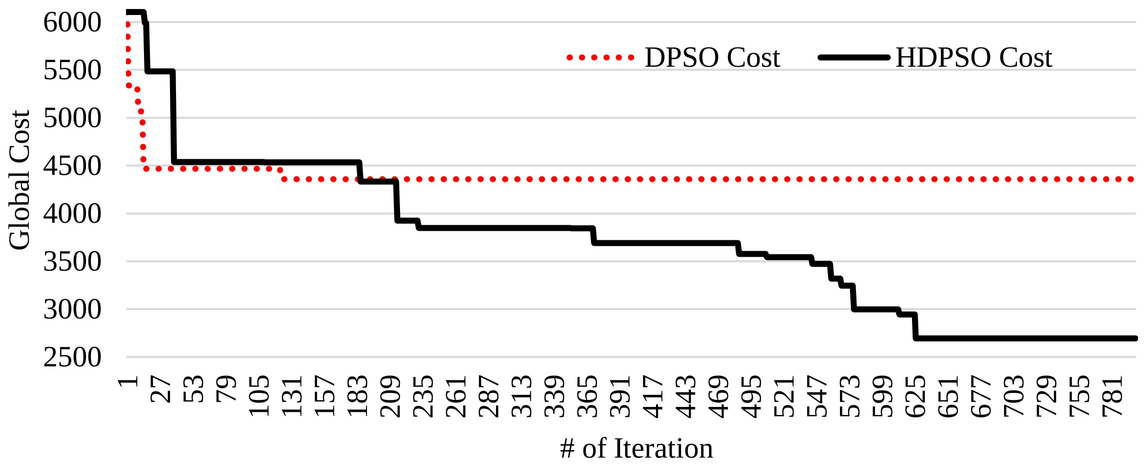


Figure 3.14 Global cost for same confidence - DPSO *vs.* HDPSO.

3.3.8 Dynamic Tuning Parameters

Self-confidence can be a static constant or dynamically calculated variable. In the case of static, the particle movement core can access this variable from a register and move on ahead with the calculation of velocity and acceleration. On the other hand, if the self-confidence needs to be dynamically calculated at run-time, it can depend on the global cost and the number of iterations. Some researchers have found a constant self-confidence by trial-and-error [27]. This means that the particle moves to local maxima with a probability of 4%. However, the self- confidence can be dynamically tuned over the course of optimization iterations by using the following equation:

$$C_{self} = \psi\left(e^{-\frac{I_{curr} \times \emptyset}{T_{iter.}}}\right) \times \varrho\left(\frac{c_{local} - c_{global}}{c_{local}}\right) \quad (8)$$

where $T_{iter.}$ is the total number of iterations to be executed

I_{curr} is the current iteration number

$\emptyset$ is the plummeting factor

ψ is designer chosen factor which determines the dependence of self-confidence on number of iteration

ϱ is a designer's chosen factor, which determines the dependence of self-confidence on particle's cost quality

c_{local} and c_{global} are is the local and global best costs respectively

For example, $T_{iter.} = 50$, $\emptyset = 4$, $\psi = 1$, and $\varrho = 0$ gives us:

$$C_{self} = e^{-\frac{I_{curr} \times 4}{50}} \quad (9)$$

It can be observed from the above equations that the data dependency on global variables increases when dynamic self-confidence values are to be calculated at runtime. However, if static self-confidence values are used, the data dependency is reduced and each particle movement can perform faster calculations.

The dynamic self-confidence can be used to influence (particles) search-agents to depend more and more on global optima as the algorithm approach to its conclusion. This effect is similar to the one used in Simulated Annealing where randomly exploring the search space is encouraged in the beginning but discouraged near the end of the algorithm. As the algorithm progresses, the particles are given less and less self-confidence so that the algorithm moves towards the global maxima. The equation can also be modified to give more and more overall swarm confidence as the iterations progress. We proposed two different ways to formulate dynamic self-confidence and dynamic swarm confidence.

3.3.8.1 Inverse exponential self-confidence and exponential swarm confidence

In this formulation, the self-confidence and swarm confidence only depend on the current number of iteration as shown in Figure 3.15.

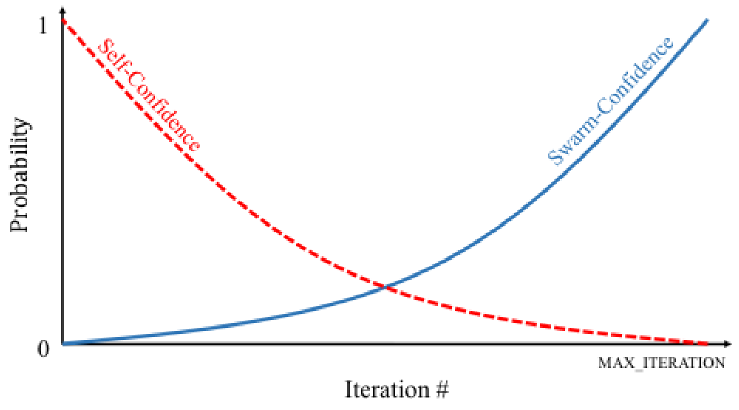


Figure 3.15 Inverse exponential self-confidence and exponential swarm-confidence.

3.3.8.2 Inverse exponential self-confidence and remaining swarm confidence

Self-confidence depends on the current iteration number where swarm confidence depends on the difference between 100% and self-confidence as shown in Figure 13.

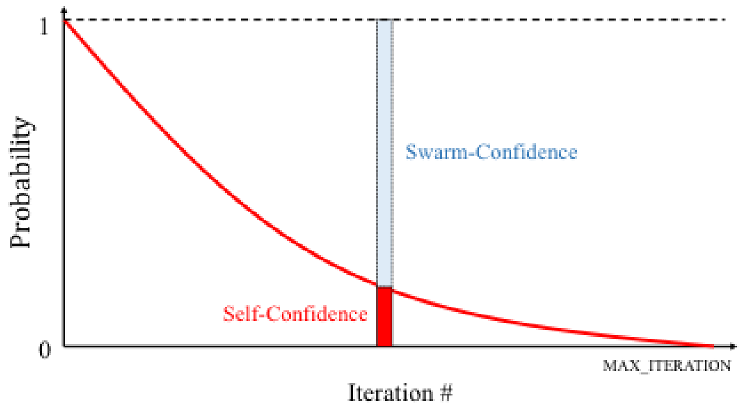
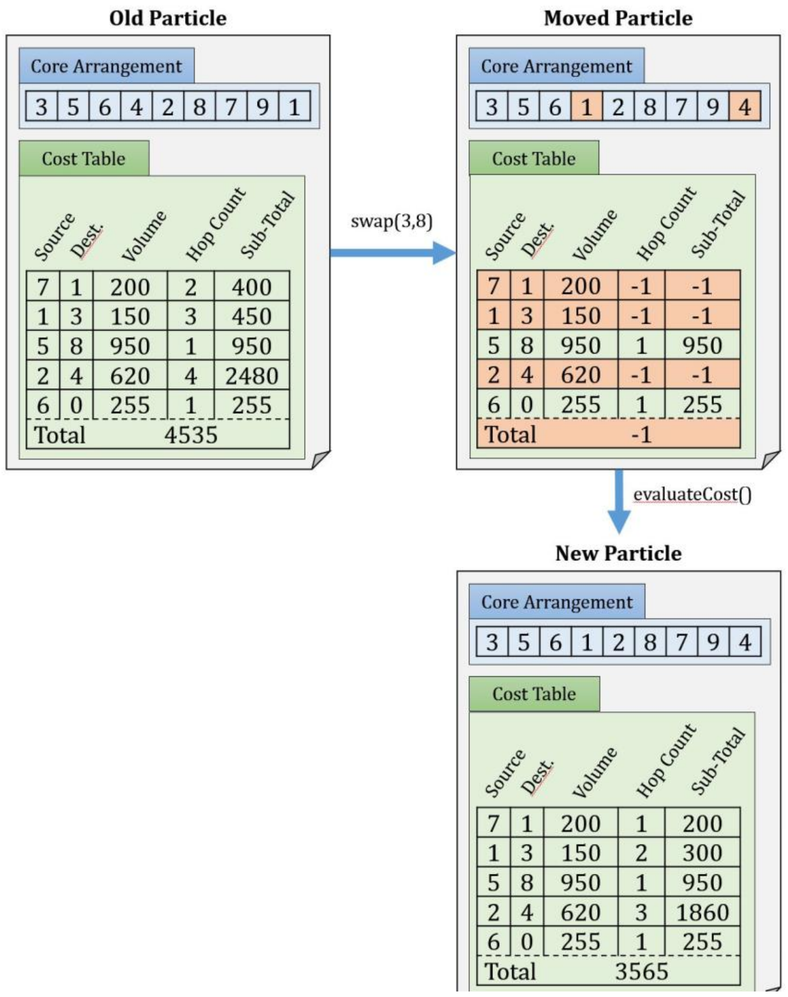


Figure 3.16 Inverse exponential self-confidence and remaining swarm-confidence.

3.3.9 Particle Cost Pre-calculation

When a particle moves to a new solution, changed edges are marked and only those costs are recalculated rather than calculating costs for entire coregraph again. This saves some computational time and makes the HDPSO algorithm faster compared to other DPSO algorithm. The mechanism is depicted in Figure 3.17.



Old Particle

Core Arrangement

3	5	6	4	2	8	7	9	1

Cost Table

Source	Dest.	Volume	Hop Count	Sub-Total
7	1	200	2	400
1	3	150	3	450
5	8	950	1	950
2	4	620	4	2480
6	0	255	1	255
Total		4535		

Moved Particle

Core Arrangement

3	5	6	1	2	8	7	9	4

Cost Table

Source	Dest.	Volume	Hop Count	Sub-Total
7	1	200	-1	-1
1	3	150	-1	-1
5	8	950	1	950
2	4	620	-1	-1
6	0	255	1	255
Total			-1	

New Particle

Core Arrangement

3	5	6	1	2	8	7	9	4

Cost Table

Source	Dest.	Volume	Hop Count	Sub-Total
7	1	200	1	200
1	3	150	2	300
5	8	950	1	950
2	4	620	3	1860
6	0	255	1	255
Total			3565	

Figure 3.17 Particle cost pre-calculation.

3.4 Experimental Results for NoC Mapping (Particles)

We have implemented a traditional DPSO and our HDPSO methodologies by using the formulation presented in this work. Both methodologies are employed to map synthetic and real application core graphs. by employing dual core i7 CPU running at 2.4 GHz having 8GB memory. Picture in Picture (PIP), MPEG4 Decoder (MPEG4), Video Object Plane Decoder (VOPD), and Dual Video Object Plane Decoder (DVOPD) are some of the typical core graphs used earlier to evaluate NoC mapping optimization [4]. Table 3.1 lists some of the previously employed benchmark core-graphs that are also used for comparison. The overall communication cost is determined for our HDPSO methodology to compare with some past NoC mapping techniques and the results are presented in Figure 3.18.

Table 3.1 Application core-graphs used as benchmark

Name	Cores	Application Modes	Edges
PIP	8	1	8
MPEG4	12	1	13
VOPD	16	1	15
DVOPD	26	1	32
DVOPD + MPEG4	38	2	45
Core Graph 1	17	3	52
Core Graph 2	29	4	92
Core Graph 3	39	5	205

In the graphical bars presentation of Figure Figure 3.18, the communication cost is normalized with the NMAP NoC mapping methodology [23] widely referred by researchers. HDPSO performs similar or better than all the mapping techniques including DPSO [24], GMAP [20], LMAP [40] and PSMAP [41]. Figure 14 shows the communication cost for VOPD, MPEG4 and PIP application mapping using HDPSO, GMAP, DPSO, PSMAP and LMPAP methods. HDPSO performs better or comparable in terms of communication cost. It is noticed that HDPSO does tend to increase time to determine the mapping solution but it reaches a stable global cost twice as fast as the standard DPSO.

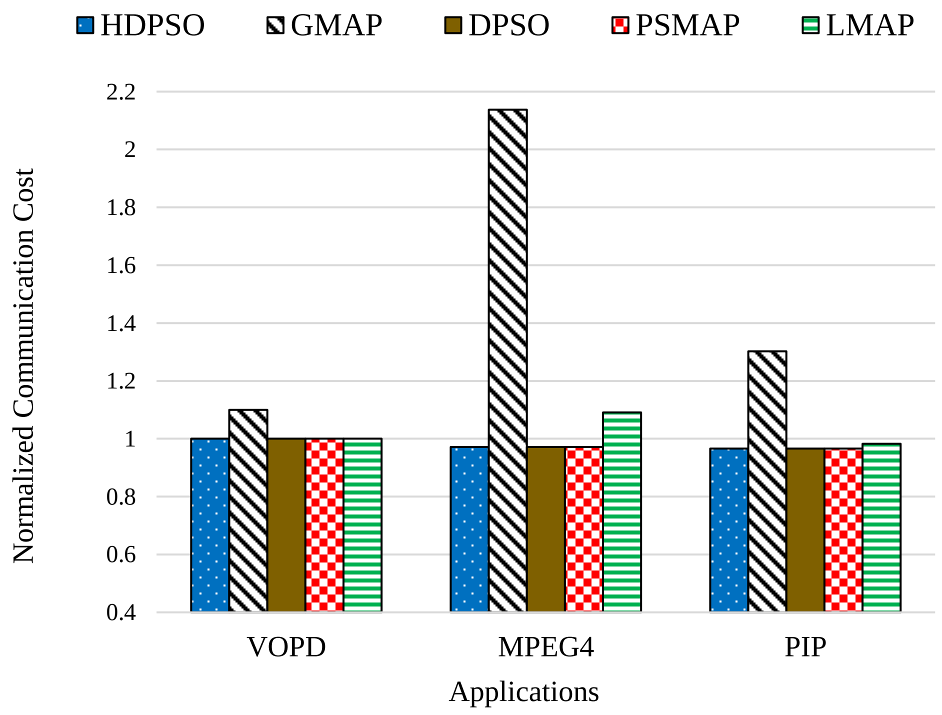


Figure 3.18 Communication cost normalized to NMAP cost.

We have implemented the DPSO-based mapping algorithm proposed by Sahu *et al.* [24, 42] to thoroughly evaluate and compare our HDPSO methodology. The NoC mappings are generated by using DPSO and HDPSO techniques for some well-known application core-graphs given in Table 3.1. Additionally, several synthetic application task graphs Core Graph 1, 2 and 3 are generated with random (non-uniform) traffic by using our custom core-graph generator that incorporate TGFF (Task Graphs For Free) pseudo-random task graph generator tools [43]. Uniform random traffic is not useful for evaluating the application mapping quality as any mapping will result in the same cost.

TGFF generates pseudo-random task graphs which can be taken as coregraphs if we assume that only one task is mapped to a single core. TGFF also generates hard and soft deadlines but they are ignored when parsing taskgraphs to coregraphs. Simple random task graphs were generated from the tgff library and converted to xml format. The communication volume was kept in between 30 and 900 Mbits/s. TGFF also takes in parameter *tg_cnt* which is basically to control the number of task graphs to generate per application. We assume each task graph as a separate application mode. Additionally, the minimum number of tasks per task graph (average, multiplier) is kept at 0.4. This means that at least 40% of the cores are used in all application modes.

NoC mapping results for all the typical and synthetic applications are presented in Figure 3.18 and Figure 3.19. Our HDPSO method reports similar or better optimal cost as compared to DPSO and other past techniques. The results presented in Figure 3.19 indicate a cost improvement for our HDPSO method with respect to DPSO. It is observed that generally for small and single mode applications, HDPSO and DPSO perform equally well in optimization. However, as the number of cores, edges, and modes increase, HDPSO produces 10% to 60% better communication cost (lower BW * Hop Count).

DPSO based NoC mapping tool produces comparable NoC mapping for PIP, MPEG4, VOPD, and DVOPD application coregraphs as these applications has only 8-to-26 coregraphs. However, HDPSO performs better for DVOPD+MPEG4 (mode-2) and synthetic application coregraphs having 17-39 cores (see Figure 15). Therefore, we have generated larger synthetic coregraphs, by using some well-known and popular realistic traffic patterns (e.g. bit-reversal, Tornado, Transpose, Shuffle, Stencil, etc.) having 32, 64 and 128 cores. Then these synthetic coregraphs are used for NoC mapping to compare our HDPSO with the latest DPSO-based method. These synthetic traffic patterns contain characteristics and features which are commonly present in real world applications. For example, communicating pairs in Bit-reversal, Tornado and Transpose traffic, ring communication pattern in Neighbor and Shuffle traffic, and dimension/array arithmetic in Stencil.

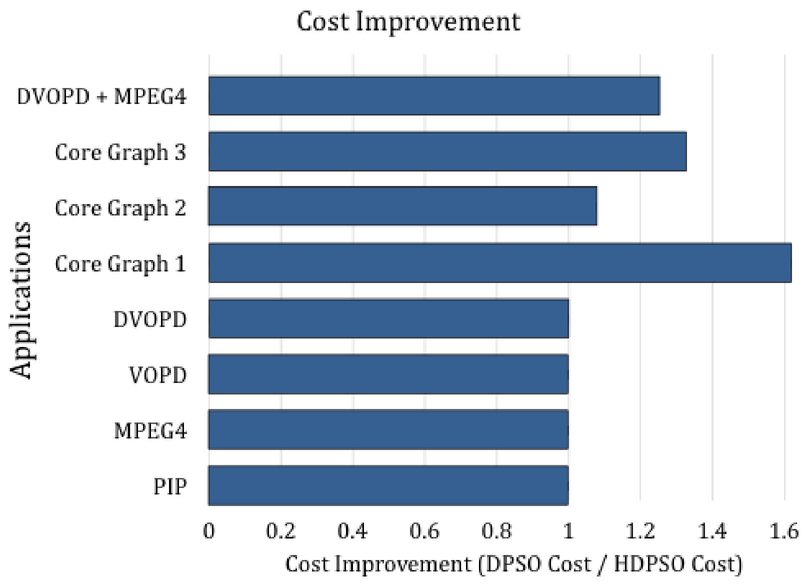


Figure 3.19 Cost improvement DPSO *vs.* HDPSO.

In Table 3.2, execution times of DPSO and HDPSO are compared for the applications given in table 1. It was noted that HDPSO found better mapping solutions for DVOPD+MPEG4, CoreGraph 1, CoreGraph 2, and CoreGraph3 at the cost of memory overhead and 5-11% more time. It did not find better solutions than DPSO in case of PIP, MPEG4, VOPD, and DVOPD because the solution found by both methods (HDPSO and DPSO) in these applications is already MILP optimal. There is no better solution to find for these applications.

For our results in Table 3.2, the number of particles was equal to the number of tiles. Therefore, as the size of application grew, the demand for tabu-ed memory grew non-linearly (ie. with a power of 2).

It was found empirically that a short-term memory which remembers just 60 past iterations is enough to discourage particles to oscillate and go back and forth between already explored solutions. However, integer is not needed for applications with number of cores less than 256 (8-bit char will do). To support larger coregraphs, our tool keeps the core ids in integer data format and hence we use that estimation. Application with 32, 64, and 128 cores will have tabu memory requirement of 311.04 Kbytes, 983.04 Kbytes, and 4976.64 Kbytes respectively.

For example, tabu-ed memory size in PIP can be calculated by $9\times 9\times 32\times 60 = 155{,}520\ Bits$ and for Core Graph 3 it can be calculated by $49\times 49\times 32\times 60 = 4{,}609{,}920\ Bits$. Network for the memory size calculation is assumed to be a square mesh. Therefore, for Core Graph 3, 39 cores fit into 7 by 7 (49 core) mesh NoC.

Table 3.2 Execution Time Comparison of Benchmark CoreGraphs

Name	Method	Time (min)	Tabu-ed Memory (Kbytes)
PIP	DPSO	0.3960	N/A
	HDPSO	0.4263	19.44
MPEG4	DPSO	0.3509	N/A
	HDPSO	0.4915	61.44
VOPD	DPSO	0.3318	N/A
	HDPSO	0.4193	61.44
DVOPD	DPSO	0.7442	N/A
	HDPSO	0.7678	311.04
DVOPD + MPEG4	DPSO	1.1963	N/A
	HDPSO	1.2023	576.24
Core Graph 1	DPSO	0.3305	N/A
	HDPSO	0.3466	150.00
Core Graph 2	DPSO	0.4091	N/A
	HDPSO	0.4490	311.04
Core Graph 3	DPSO	0.8165	N/A
	HDPSO	0.8949	576.24

We also built a tool to generate synthetic traffic patterns using well-known traffic formulae called CoreGraph generator (CGG). The coregraphs given in Table 3.3 are generated by CGG. CGG creates a coregraph xml file ready for consumption into our HDPSO tool. It takes in the traffic pattern option and size of coregraph to generate as parameters. Then iterates through each source edge, creates an edge originating from that source core, and applies the traffic pattern formula to determine edge's destination core. The tool assumes that the cores are mapped with respect to their ids (ie. core 0 on the top-left, core 1 next to core 0 on the right, and core N at the bottom-right). HDPSO is expected to detect these traffic patterns and map them in another recognizable pattern. For example, communicating pairs in transpose and bit-reversal traffic are placed at 1 hop distance in an optimal solution after tool run. This helps evaluate the tool and helps in identifying tuning parameters.

Synthetic traffic can be divided into two major categories named random and permutation. Random (uniform) is where each core sends the same amount of data to every other core in the network. From mapping perspective, uniform random traffic is not useful for evaluating the mapping tool as any mapping will result in the same cost. Random (non-uniform) is where each core sends data to one or several random destinations. Random (non-uniform) traffic pattern is used in the synthetic core graphs (Coregraph 1, Coregraph 2 and Coregraph 3) employed earlier and results for those were presented in Figure 3.18. Permutation is where the destination address is a permutation function of source address. To generate permutation traffic, each core in the network is assigned a binary address and then the permutation function is applied to each core (source) to produce resulting destination address. Each source sends data only to one destination. There are two sub sets of permutation i.e. bit and digit permutation. In bit permutation traffic, the destination core address is computed by permutating or selectively complimenting the bits of the source address. Digit permutations are done by permutating digits of source address to calculate destination digits. Digit permutations can only be applied to n-digit, radix-k numbers for example k-ary n-mesh networks.

Bit reversal traffic is shown in Figure 3.20 (a) and it is generated by reversing the order of source address bits to generate destination address. It generates pairs of bi-directionally communicating cores which the mapping tool should place near each other. A traffic type which best tests locality exploitation of the mapping tool is the neighbor traffic shown in Figure 3.20 (b). In this type of traffic, each source communicates with a core in the next column and next row. The optimum solution reduces majority of the hop count from 2 to 1 of each neighboring core pair. Tornado traffic is shown in Figure 3.20 (c) that is designed to test the mesh and torus networks. The optimum solution finds pairs in the network and places them in proximity. The Transpose traffic is shown in Figure 3.20 (d) where source cores talk to the core on the other side of the diagonal line. It creates pairs of communicating cores and the optimum mapping should place these pairs near each other.

Fast Fourier Transforms (FFTs) and sorting applications have shuffle type traffic patterns, which is shown in Figure 3.20 (e). There are communication localities in the traffic, which need that the relevant cores should be placed in proximity to each other. For example, in Figure 3.20 (e), some cores form a communicating ring and they need to be placed in a 2×2 mesh square in the network to have communicating cores just 1 hop count away from each other. Signal and image processing applications often have traffic patterns that are known as Stencil shown in Figure 3.20 (f). Some relevant applications include FIR filtering, 2D convolution.

We have investigated the conventional DPSO mapping technique [5, 18] along with our HDPSO methodology for the synthetic traffic patterns shown in Figure 3.20 and for 32, 64 and 128 core applications. The comparative results for all the synthetic core graphs with various traffic pattern are presented in Table 3.3. Our HDPSO based mapping shows considerable improvement in cost for large networks. The cost improvement ranges from 15% to 216%. Results for the Neighbor, Tornado and Shuffle traffic patterns are particularly better (i.e. 23% to 216% improvement) as these traffic patterns contain ring communication structure which can easily be exploited by traffic optimization stage of HDPSO method (i.e. stage D described in Section 3.3.6.4).

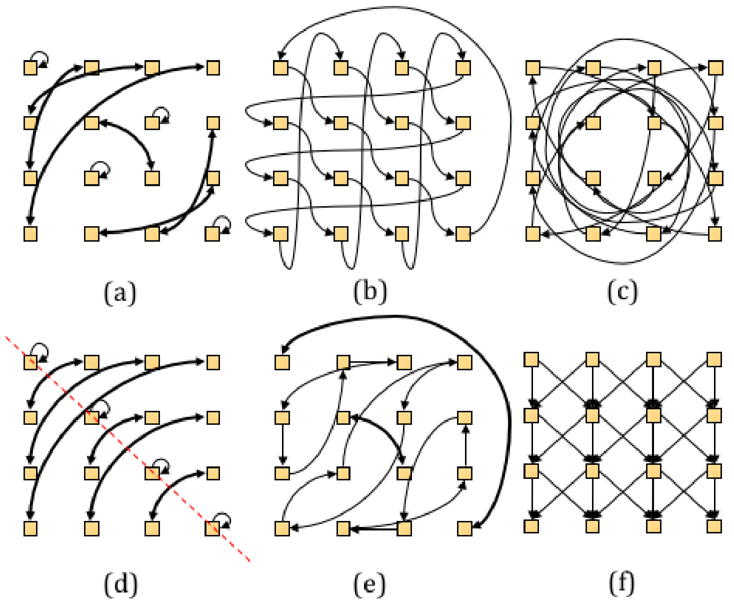


Figure 3.20 Traffic patterns used to evaluate the HDPSO based NoC mapping (a) Bit-reversal, (b) Neighbor (c) Tornado (d) Transpose (e) Shuffle (f) Stencil.

It is also observed that the DPSO method settled relatively quickly (in fewer number of iterations) on the final solution as compared to our HDPSO. Even though HDPSO employs Tabu-list and needs to check for Tabu-ed moves in each iteration, which should increase the time taken per iteration, DPSO and HDPSO still have comparable execution times to find an optimal solution. The main reason is the capability of our HDPSO method that remembers the previous iteration costs of each particle and if the particle does not move, the cost is not recalculated while DPSO recalculates the cost in each iteration. HDPSO methodology also consumes more memory than DPSO because it remembers the tried core arrangements (or swarm particles data) for 60 iterations. However, HDPSO takes less number of overall iterations to reach an optimal solution since it looks for more regions of the search space and progresses using multiple (1→4) swap moves per iteration.

Like all the stochastic techniques, tuning parameters play a major role in how well the optimization tool performs for a given input. We have used static confidence values (tuning parameters) to generate all the results in this section. It was found that particle inertia of 0.4, self-confidence of 0.2, swarm confidence of 0.3, and load optimization constant of 0.4 works best in reaching an optimal solution much faster. In this way, the confidence values are empirically chosen. It can be observed from the results presented in Table 3 that the tuning parameters we chose empirically, usually favor NoC mapping to meshes that contain 64 cores because the improvement is better for those and then it bounces back for larger networks.

The HDPSO based NoC mapping tool also automatically selects a 2D-mesh size that is based on the number of cores of the application core-graph, however, the mesh size can also be selected manually. In the case of MPEG4 decoder, the mesh size is automatically selected as $4\text{x}4$, but the application can also fit to a $3\text{x}4$ size mesh NoC. The NoC mappings generated by HDPSO methodology is shown in Figure 17 for $3\text{x}4$ mesh. The communication costs for MPEG4 mapped onto a $3\text{x}4$ 2D mesh is 3772 as compared to 3567 for a $4\text{x}4$ mesh mapping shown in Figure 3.21.

In Figure 3.20, it can be observed that the bit reversal, transpose, and shuffle traffic patterns have bi-directional edges. One bi-directional edge is counted as 2 directional edges in Table 3.3. For example, in case of bit-reversal, the reverse of core 1 ie. 0b 00001 is core 16 ie. 0b 10000 and vice versa. One edge is from core 1 to 16 and another edge is from core 16 to 1. If a core’s id is palindrome, it does not have an edge. There are total 8 palindromes in 5-bit addresses (32 cores see table 3 bit-reversal). Therefore, there are $32 - 8 = 24$ edges in 32-core bit-reversal application.

For results in Table 3.3, it was observed that HDPSO and DPSO both settled on a solution after 1000 iterations on average. The times reported in the table are for 2000 iterations.

Table 3.3 Traditional DPSO vs HDPSO results for 32, 64, and 128 core synthetic application core-graphs

Traffic	Cores	Edges	Method	Time Taken (min)	Cost	Tabu Memory (KBytes)
Bit Reversal	32	24	DPSO	0.41	3000	311.04
			HDPSO	0.41	2600	
	64	56	DPSO	2.53	32000	983.04
			HDPSO	2.55	20000	
	128	112	DPSO	35.23	116400	4976.64
			HDPSO	36.55	75600	
Neighbor	32	31	DPSO	0.71	6900	311.04
			HDPSO	0.72	5600	
	64	64	DPSO	6.20	22800	983.04
			HDPSO	6.22	17000	
	128	127	DPSO	106.06	76200	4976.64
			HDPSO	107.28	59200	
Tornado	36	36	DPSO	0.52	5600	311.04
			HDPSO	0.52	4000	
	64	64	DPSO	4.23	18400	983.04
			HDPSO	4.25	12800	
	128	128	DPSO	76.40	142400	4976.64
			HDPSO	74.32	45000	
Shuffle	32	32	DPSO	0.45	7000	311.04
			HDPSO	0.43	5200	
	64	64	DPSO	4.35	44800	983.04
			HDPSO	4.37	17400	
	128	128	DPSO	7.66	95800	4976.64
			HDPSO	7.82	60800	
Transpose	32	30	DPSO	0.44	6000	311.04
			HDPSO	0.45	5000	
	64	56	DPSO	4.33	17200	983.04
			HDPSO	4.30	10800	
	128	126	DPSO	5.29	78200	4976.64
			HDPSO	4.99	60200	
Stencil	32	69	DPSO	0.50	34800	311.04
			HDPSO	0.48	27200	
	64	154	DPSO	4.86	129200	983.04
			HDPSO	4.85	88000	
	128	329	DPSO	111.95	445800	4976.64
			HDPSO	112.08	366800	

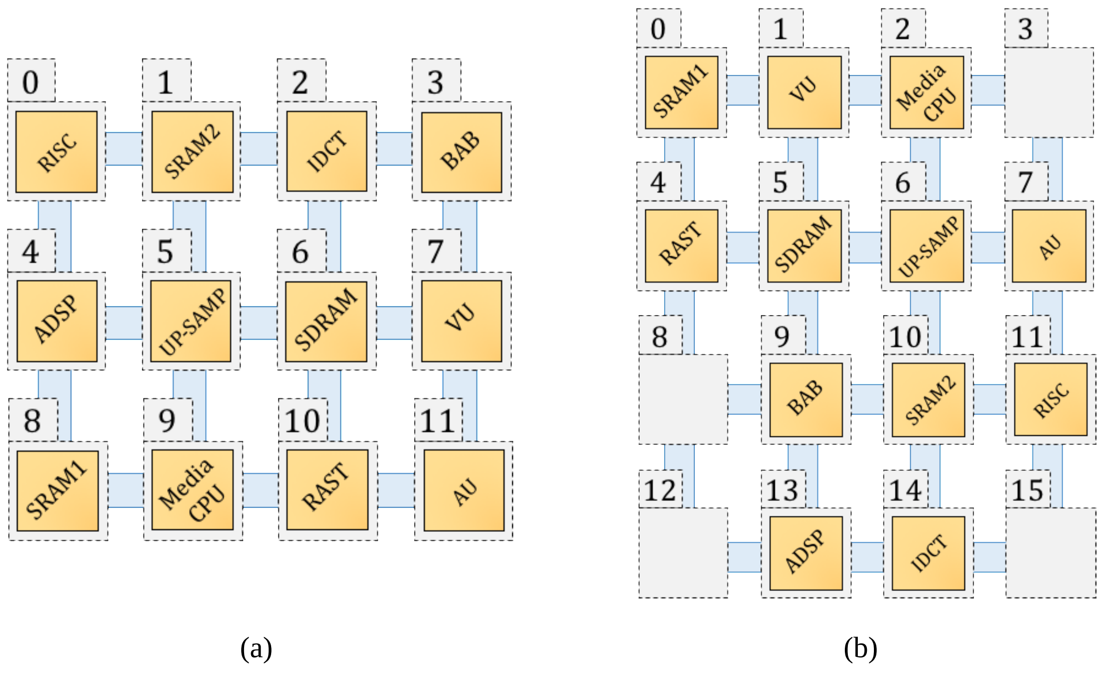


Figure 3.21 HDPSO generated optimal (a) 3×4 (b) 4×4 NoC mapping of MPEG4 decoder.

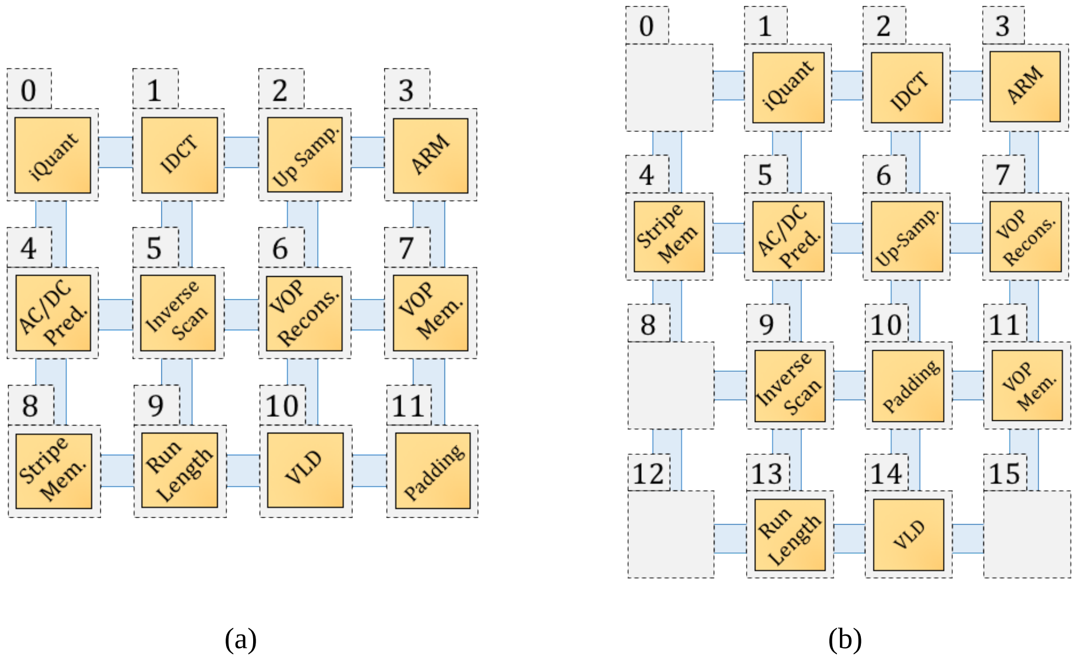


Figure 3.22 HDPSO generated optimal (a) 3×4 (b) 4×4 NoC mapping of VOP decoder.

3.5 Summary

A new method of optimization was discussed which increases the efficiency of the traditional DPSO algorithm by tabu-ing parts of search areas to particles which have been just visited. Since the search space was of combinatorial discrete nature, deflection technique was used to encourage particles to explore more of search space before converging to the optimal solution. If search space was of continuous nature, other methods such as Bezier curve would be used to force the particle to curve around the tabu-ed area.

If a swap move results in a solution which is already tried, proposed moves are deflected in all other directions until there is no other way other than to allow the original move. This causes the algorithm to explore more of the search space before converging onto the global best solution. The results were benchmarked against popular applications PIP, MPEG-4, VOPD, and DVOPD and it was found that HDPSO performs significantly better than other optimization algorithms. Standard DPSO based method has also been implemented for NoC mapping and quality of the results is compared with our HDPSO based mapping. It was found that HDPSO performs 15% to 216% better than traditional DPSO technique. Our tool also performed better in particular for traffic patterns which contain ring communication structure such as Neighbor, Tornado, and Shuffle traffic patterns. It is also concluded that remembering the past solutions and proposing new solutions by our HDPSO algorithm enables it to produce better mapping solutions.

Chapter 4

Optimal NoC Configuration using Sub-swarm

4.1 Introduction

In addition to NoC mapping, next major steps in designing NoC include finding paths for the traffic flows while reserving resources across the NoC and determining NoC architectural parameters, such as the data width of the links, buffer sizes used in the router, and frequency of operation [6]. After cores are mapped to their respective nodes in topology, each communication edge in the core graph has to be assigned a physical path (links) with a bandwidth high-enough to support it. Then the resources along these paths are reserved to support edges communicating through them. Resources should be allocated generously to be able to support the collective bandwidth required by the edges communicating along the path, but at the same time to have minimum chip area and power consumption. These conflicting requirements give rise to the configuration optimization of NoC. Since there are many different possibilities (combinations) of architectural parameters that can satisfy a given application bandwidth on a given topology, there is a need to find an optimal solution. On the same lines of NoC mapping, NoC configuration solution space also grows factorially with an increase in the number of cores and library options available. Therefore, there is a need to employ a stochastic technique to consider promising solutions rather than developing a deterministic methodology.

We propose a Hybrid Discrete Particle Swarm Optimization (HDPSO) based tool that takes an Application Core Graph (ACG) and network component libraries and provides an optimal application specific NoC design. Our proposed NoC design tool uses stochastic searching techniques to first generate random particles (solutions) called initial heuristic particles and then iteratively moves the particles around the search space to explore other solutions.

Our main contributions include the development of an intelligent memory-based hybrid particle multi-swarm optimization technique which combines exploration abilities of particle swarm with exploitation abilities of force directed swapping. Additionally, we introduce multiple swarms to search configurations and mapping search space separately. We assume a library of synthesized router sub-components is provided. We have developed a new technique by combining the NoC mapping optimization problem with the architectural configuration for a cooperative solution search.

The overall architecture of our proposed system is shown in Figure 4.1. The Application Communication Graph consisting of a core graph for each application along with communication cost model, area library, and power library is fed to the system. Finally, the system produces a core-to-tile map, link configurations, router configurations, optimal NoC area, power, and communication cost. Core-to-tile map points to where a particular core is to be located in the 2D-mesh. The proposed tool explores to provide the best configuration which results in optimal communication cost, power, and chip area.

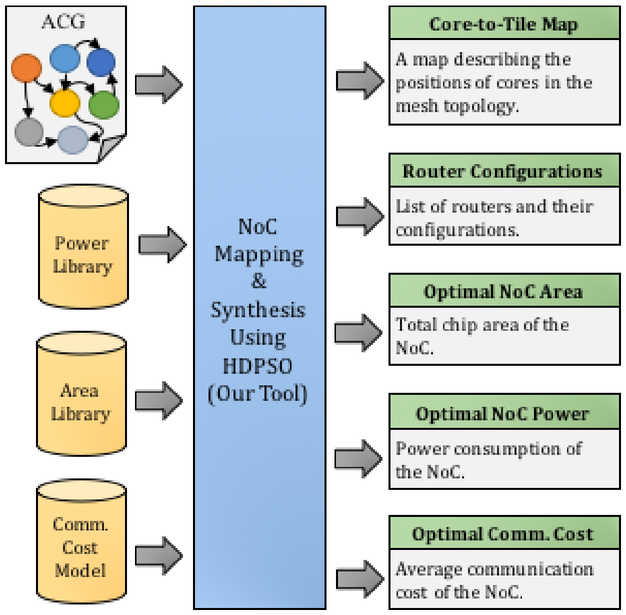

ACG
Power Library
Area Library
Comm. Cost Model
NoC Mapping & Synthesis Using HDPSO (Our Tool)
Core-to-Tile Map
A map describing the positions of cores in the mesh topology.
Router Configurations
List of routers and their configurations.
Optimal NoC Area
Total chip area of the NoC.
Optimal NoC Power
Power consumption of the NoC.
Optimal Comm. Cost
Average communication cost of the NoC.

Figure 4.1 CAD system architecture for NoC synthesis

The resulting design contains description of routers, topology graph, and link configurations. Topology graph consists of link IDs, router IDs and their connections. Each link configuration consists of link width and maximum virtual channels (VCs) operating on the link. Router configuration comprises of SRAM buffer depth (in case of dynamic VC organization), VC buffer depth (in case of static VC organization), type of arbiter used in each router, arbiter size, crossbar switch size, type of flow control, and estimated router area and power consumption. A router (by extension a NoC) can be designed by choosing a flow control technique, the number of VCs for each input port, buffer organization, switch design, and pipelining strategy while abiding by the target clock frequency and power budgets. The buffer size, microarchitecture design and number of VCs for the router can be determined for the router when provided with the network topology, application graph, some architecture specific parameters, and probability distributions of packet properties. These parameters must be designed in such a way that optimal performance is achieved at minimal area and power costs.

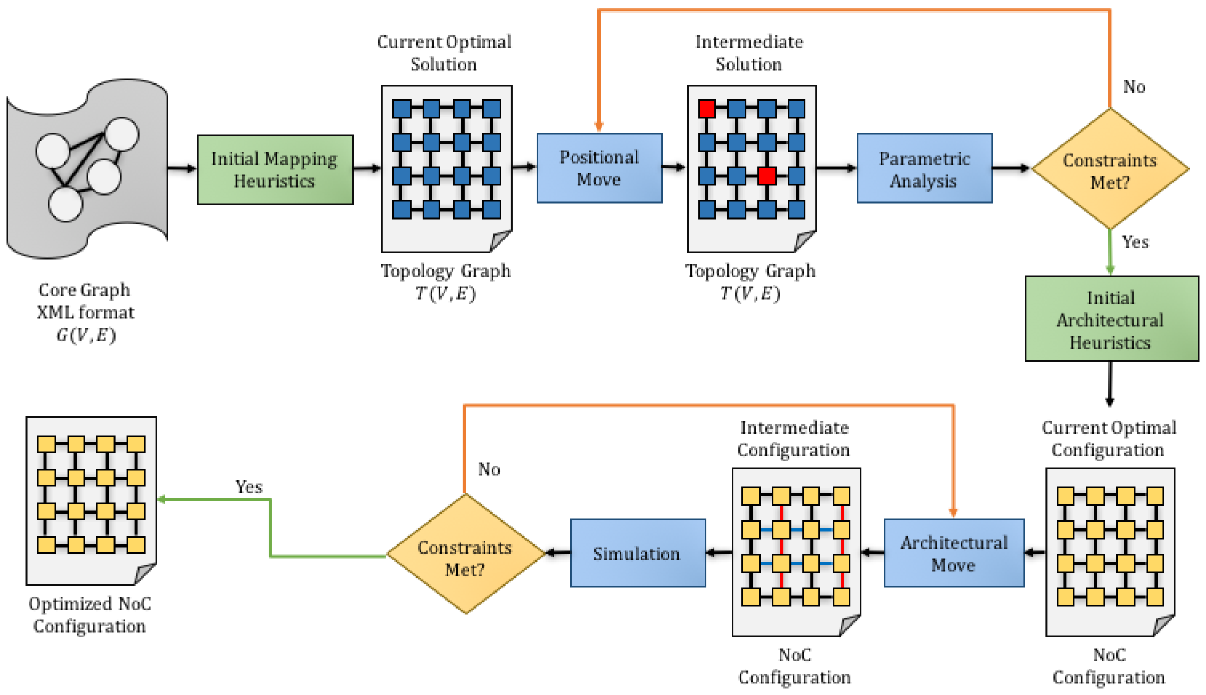
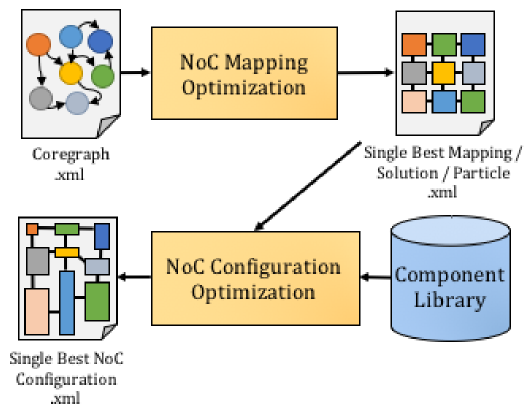
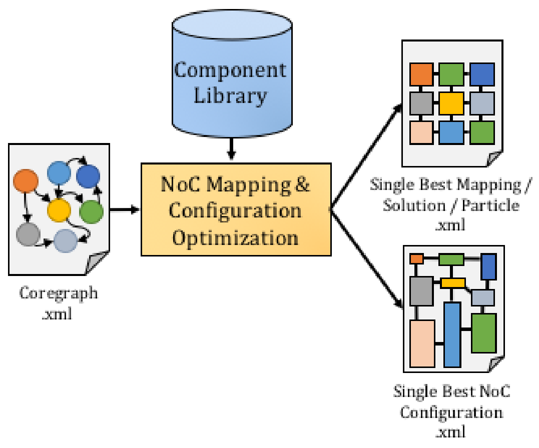


Figure 4.2 Typical NoC synthesis workflow.

Generally, NoC mapping and configuration are categorized as ‘segregated search space optimization’ and ‘combined search space optimization’ as depicted in Figure 4.3a and Figure 4.3b respectively. Segregated search space NoC optimization is the most commonly used and tend to first generate an optimal NoC mapping based on power, area, throughput, etc. and then choose the network components to work optimally with the mapping based on the network performance. However, such a segregated search space approach has several disadvantages. The results obtained from the mapping stage might no longer be optimal after network configuration. We believe that mapping and configuration optimization problems of NoC are closely related to each other. An optimal result must be picked after performance verification of mapping and configuration solution pairs.

Æthereal performs application specific NoC synthesis as depicted in Figure 4.3a [8]. It has two optimization loops of buffer sizing and smallest mesh. Smallest mesh loop is responsible for generating a topology and a mapping for the given ACG. The buffer-sizing loop resizes the buffers in each router for the generated NoC map and topology. Combining the search spaces of mapping and configuration together and looking for optimal mapping/configuration as one optimal solution may find more optimal solutions rather than the segregated solution spaces. Yet, this approach widens the scope of search space and may result in spending time in finding configurations for non-promising mappings. Regardless of the cost of mapping, its configuration is found. Moreover, many explored solutions might not meet the design constraints and get rejected during performance verification stage after wasting lot of time.


Coregraph .xml
NoC Mapping Optimization
Single Best Mapping / Solution / Particle .xml
NoC Configuration Optimization
Component Library
Single Best NoC Configuration .xml


Component Library
Coregraph .xml
NoC Mapping & Configuration Optimization
Single Best Mapping / Solution / Particle .xml
Single Best NoC Configuration .xml

(a) (b)

Figure 4.3 (a) Segregated search space optimization workflow (b) combined search space optimization.

We propose to combine NoC mapping and configuration optimization problems by use of swarms and sub-swarms. This will obviously cause bigger search space dimensions than pure mapping or pure synthesis search space, but should result in more optimal final solution as explained earlier. This kind of approach provides us with two advantages over traditional swarm optimization. It can be observed that the communication cost of a candidate solution depends on the communication volume of an edge which is extracted from the input core graph and hop count for that edge which is extracted from the mapping solution itself. Since, this cost can be calculated before configuration, it is a better idea to do partial calculation of particle cost (P_{Cost}) before moving to NoC configuration. This partial cost drives the main swarm and configuration cost ($Conf._{cost}$) drives the sub-swarm.

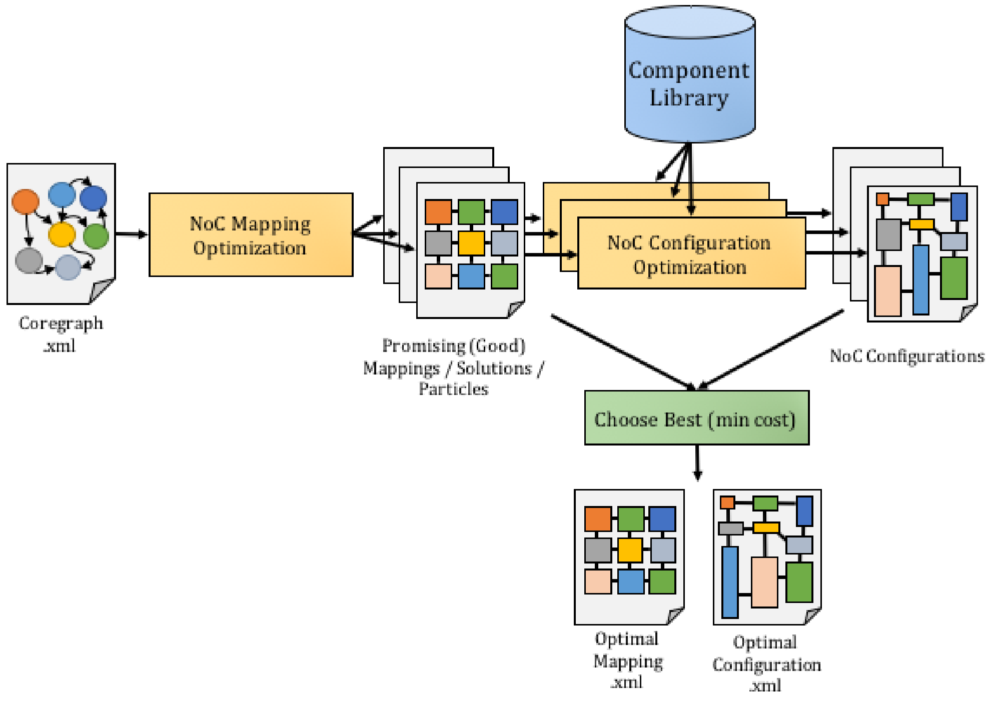


Figure 4.4 The proposed hybrid workflow for NoC mapping and configuration.

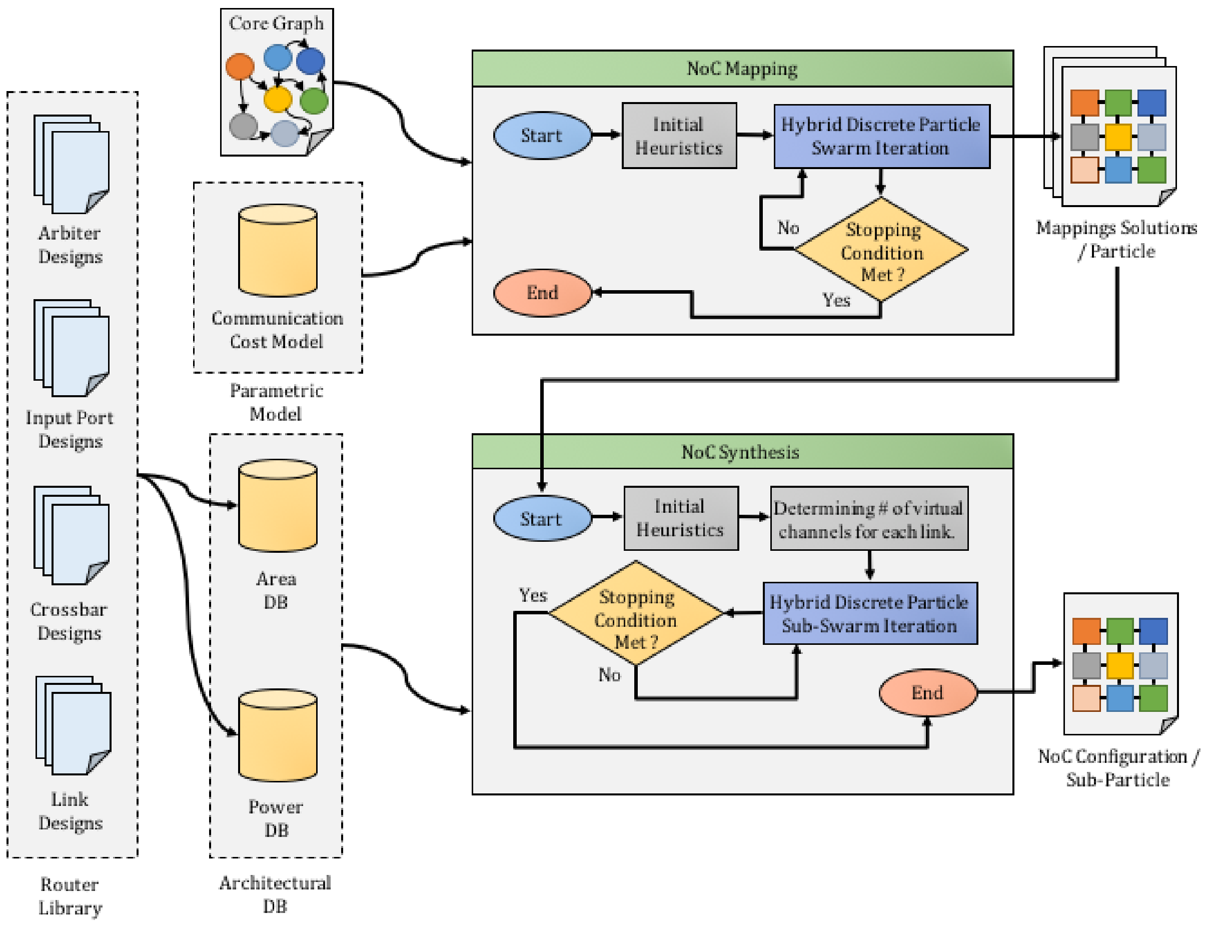


Figure 4.5 Inner workings of the mapping and configuration HDPSO algorithm.

4.2 Problem Definition

As we described earlier, when the number of NoC cores grow, the solution space explodes and there is a need for some optimization tools to find the optimal solution. An embedded application can be represented by an Application Core Graph (ACG). ACG comprises of collection of application modes ($AM \in ACG$). Where each $AM = G\{C, E\}$ is composed of collection of cores (vertices) C and collection of edges (unidirectional links) E. Moreover, each directed edge is denoted by $e_i = v_i(c_j, c_k) \in E$ where c_j is core j, c_k is core k, v_i is the maximum volume of communication that can occur between j and k cores in Mega-bits/sec. The objective is to configure the NoC in such a way that the total communication cost, power consumption, and total chip area occupied by the NoC is minimal. NoC design optimization problem is of multi-dimensional nature, where the first step is to identify the number of dimensions of the search space. There are multiple objectives of this optimization problem as given below:

- Minimize hop count between frequently communicating cores across one or multiple mode application.
- Minimize static and dynamic power consumption of NoC routers and links.
- Minimize the chip area occupied by routers and links.
- Minimize buffer penalty
- Minimize credit loop latency
- Maximize duty factor on links

We assume a regular mesh topology with XY routing mechanism. We also assume that Virtual Channel (VC) buffers are part of the input ports of the router. We define the cost for the configuration solution or sub-particle ($Config._{cost}$) as:

$$Config._{cost} = \left(\alpha P_{Cost} + \beta NoC_{power} + \gamma NoC_{area}\right).\lambda B_{penalty} \tag{10}$$

where $\alpha, \beta, \gamma, \lambda$ are tuning parameters for dependency of total cost on the particle cost (P_{cost}), power of NoC (NoC_{power}), area occupied by NoC (NoC_{area}) of NoC, and Buffer Depth Penalty ($B_{penalty}$) respectively.

Particle cost (P_{cost}) is given by:

$$P_{cost} = \sum_{j=1}^{size(AM)} \sum_{i=1}^{size(E)} Volume(e_i) \times HopCount(e_i) \tag{11}$$

where edge *i* ($e_i \in E$), $Volume(e_i)$ is the communication volume in MB/s on edge *i*, $HopCount(e_i)$ represent the hop distance between two cores.

Power cost of the NoC can be estimated by using following equation:

$$NoC_{power} = \sum_{i=1}^{size(R)} P(R_i) + \sum_{j=1}^{size(L)} P(L_j) \tag{12}$$

where R and L are the collection of all routers and links,

Router $R_i \in R$ and *Link* $L_j \in L$,

$P(L_j)$ & $P(R_i)$ are power consumption of *Link j* & *Router i*.

Since the router components can be roughly modularized into three types namely crossbar, arbiter, and input ports, the router power consumption can be found by adding the power consumption of all the router components given in the library:

$$P(R_i) = P(A_i) + P(C_i) + \sum_{k=1}^{size(I)} P(I_k) \tag{13}$$

$P(A_i)$ and $P(C_i)$ are the power consumed by the arbiter and crossbar switch of *Router i*,

I is the collection of input ports of *Router i*,

$P(I_k)$ is the power consumed by *k input port* of *Router i*.

Chip area for the solution or particle is found as following:

$$A_{NoC} = \sum_{i=1}^{size(R)} A(R_i) + \sum_{j=1}^{size(L)} A(L_j) \tag{14}$$

where R and L are the collection of all routers and links,

Router i $(R_i \in R)$ and *link j* $(L_j \in L)$,

$A(L_j)$ & $A(R_i)$ represent the chip area of *Link j* & *Router i*.

The area of a router can be determined from a library by looking into the architectural feature areas of router as follows:

$$A(R_i) = A(A_i) + A(C_i) + \sum_{k=1}^{size(I)} A(I_k) \tag{15}$$

where $A(A_i)$ and $A(C_i)$ are the chip areas of arbiter and crossbar switch of *Router i*,

I is collection of input ports and $A(I_k)$ is the area of *k* input ports of *Router i*.

When the flow of data suddenly starts after being blocked, the downstream router needs to inform the upstream router to send more data. This is done through credits. However, credits take time to travel back to the upstream router. Additionally, after receiving credits, data takes time to reach the downstream router. The total time for the credits to reach back upstream router and the time it takes for data to reach downstream data is known as credit loop latency t_{crt}. For this time, link bandwidth at the downstream router is not being utilized.

The credit loop latency t_{crt}, expressed in flit times, gives a lower bound on the number of flit buffers needed on the upstream side for the channel to operate at full bandwidth, without credit stalls [44]. Since each flit needs one buffer and the buffers cannot be recycled until their tokens traverse the credit loop, if there are fewer than t_{crt} buffers, the supply of buffers will be exhausted before the first credit is returned.

The credit loop latency t_{crt} in flit times is given by:

$$t_{crt} = t_f + t_c + 2T_w + 1 \tag{16}$$

where t_f is the flit pipeline delay, t_c is the credit pipeline delay, and T_w is the one-way wire delay (we assume this to be 1)

If $B(flow_j)$ is the number of buffers available to data flow $flow_j$ in the input port, to calculate average buffers allocated to all data flows in a link ($B_{avg.}(link_i)$)following equation is used:

$$B_{avg.}(link_i) = \frac{\sum_{j=1}^{NFlows(link\ i)} B(flow_j)}{NFlows(link\ i)} \tag{17}$$

where $NFlows(link\ i)$ is the number of data flows going through $link_i$

To calculate link duty factor (df) for $link_i$, following equation is employed:

$$df(link_i) = min\left(1\ , \frac{B_{avg.}(link_i)}{t_{crt}}\right) \tag{18}$$

The duty factor of the data flow will be 1 as long as there are enough buffered flits to hide the credit loop latency. To calculate average duty factor for a configuration solution, following equation is used:

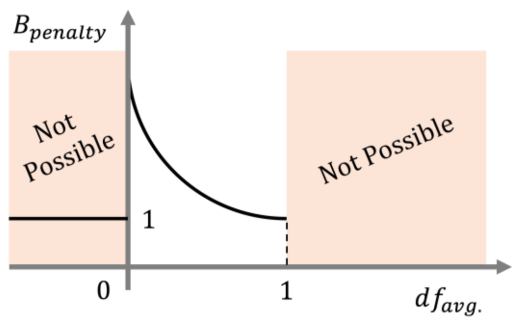
$$df_{avg.} = \frac{\sum_{i=1}^{N_{links}} df(link_i)}{N_{links}} \tag{19}$$

where N_{links} is the number of total links in the network

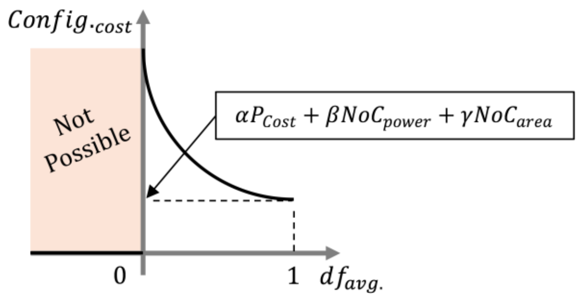
Using the duty factor (df), we determine the buffer penalty to be as following:

$$B_{penalty} = \frac{1}{df_{avg.}} \tag{20}$$

Figure 4.6 shows how buffer penalty and configuration cost are affected by average duty factor. If $df_{avg.}$ is below 1, the sum $\alpha P_{Cost} + \beta NoC_{power} + \gamma NoC_{area}$ is multiplied by inverse of $df_{avg.}$ which has huge impact on the cost. In other words, df_{avg} is inversely proportional to configuration cost.



(a)



(b)

Figure 4.6 (a) Average duty factor and buffer penalty relationship (b) Average duty factor and configuration cost relationship.

4.2.1 Constraints

Any configuration and mapping solution should not be allowed to take the link utilization above 1. Link utilization above 1 means that the link required throughput is higher than the link bandwidth. The Link Utilization is given by:

$$Link_{util.} = \frac{\sum_{i=1}^{N_f} Volume(e_i)}{L_{width} \times L_{freq}} \leq 1 \qquad (21)$$

where $\sum_{i=1}^{N_f} Volume(e_i)$ is the amount of data flowing through the link or in other words, sum of all data flows through the link

L_{width} is the width in bits of the communication link

L_{freq} is the operating frequency of the link

4.3 Configuration Solution Space

Since mapping and configuration optimization solutions are closely related to each other, we propose to search configuration search space simultaneously as mapping solution space is being searched. This is done by the mechanism of sub-swarm and sub-particles. Each particle has a sub-swarm which contains a fixed number of sub-particles. These sub-particles are possible configuration solutions. Whenever a particle moves (ie. finds new mapping solution), the sub-swarm is notified and all sub-particles are destroyed and then re-spawned. In each iteration of mapping optimization (ie. particles), several iterations of sub-particles are run. This particle-to-sub-particle iteration ratio is controlled by a parameter set before runtime.

The configuration solution space size depends on number of cores, number of router component configurations available, and mapping solutions space. The configuration solution space size is given as:

$$Conf.Sol._{size} = \left(N_{cores}.\left((5.N_{iprt.confs.}) + N_{cross.confs.} + N_{arb.confs.}\right)\right).Map.Sol._{size} \tag{22}$$

where N_{cores} is the number of cores in the coregraph

$N_{iprt.confs.}$ is the number of input port configurations in the library

$N_{cross.confs.}$ is the number of crossbar configurations in the library

$N_{arb.confs.}$ is the number of arbiter configurations in the library

$Map.Sol._{size}$ is the number of mapping solutions in solution space

As a particle encounters new mapping solutions, potential configurations solutions are tried on it by sub-particles in sub-swarm. A sub-swarm is tied to a mapping solution or particle. As particle moves around, so does the sub-swarm. By nature of DPSO algorithm, mapping particles stay in a mapping solution for a longer duration if the solution is good. This increase in time allows sub-particles to penetrate deeper into configuration solutions space and look for more solutions. All in all, more configuration solutions will be explored for good mapping solutions.

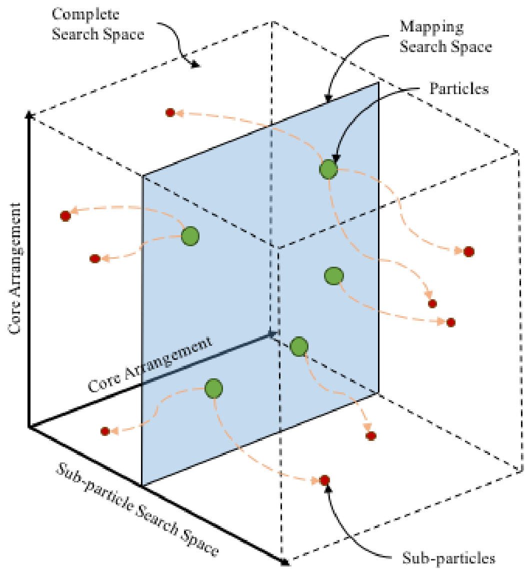


Figure 4.7 NoC Configuration solution space.

4.4 Router Sub-component Library Representation

The router components are synthesized separately using synthesis tools such as Synopsys Design Compiler and using various technology size libraries. Router component area and power parameters are converted into E**X**tensible **M**arkup **L**anguage (XML) format.

The hierarchical relationship of the elements is shown in **Error! Reference source not found.** and REF _Ref491140696 \h Figure 4.9. A router can have up to 1 component of type ***'Crossbar'*** of any available configuration. It may also have up to 1 component of type ***'Arbiter'*** of any available configuration. Finally, it may have up to 5 components of type ***'Input Port'*** of any available configuration.

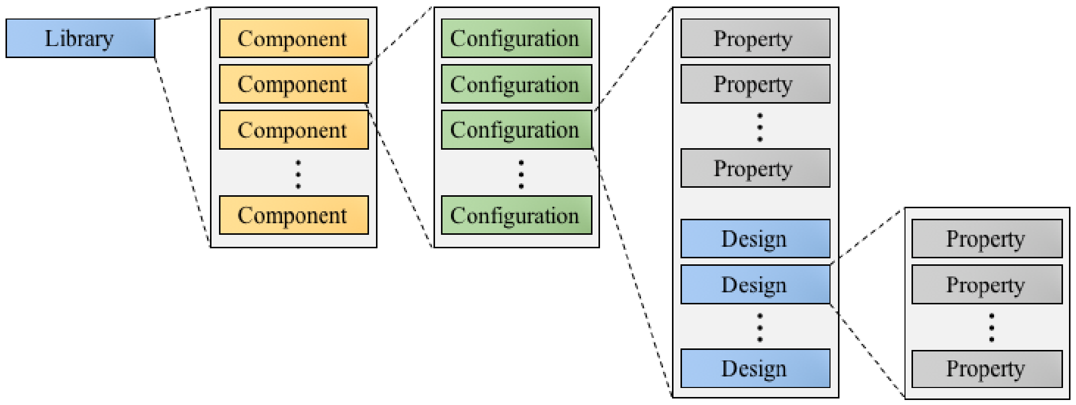


Figure 4.8 NoC Configuration library XML element hierarchy.

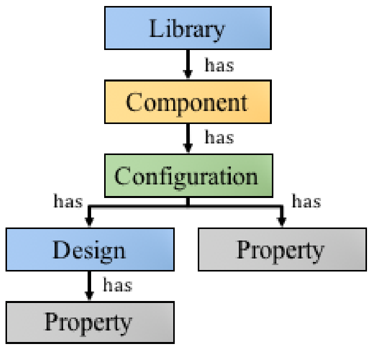


Figure 4.9 Library element hierarchy and relationship.

When any library is imported into the tool, each component and configuration pair (regardless of the component type) receives a unique id (uId). This uId is used throughout the configuration process and iterations. Each configuration solution (sub-particle) is an array of uIds. Detailed structure of a sub-particle is described in Section 4.5.

4.4.1 Library

A library is a database of components. It is represented by ***<library>*** xml element which contains several ***<component>*** elements. When the tool runs, several different libraries can be imported into the solution space using the import library button. The tool forms one library out of all the components present in libraries to be imported by use of ***merge_library*** function. This selective importing of libraries allows the designer to choose specific target libraries of router components and reduce solution space for the configuration HDPSO sub-swarm algorithm.

```
<Library>
        <Component Id="0" Name="VC-Free" Type="Input Port" Detail="…">. . . </Component>
        <Component Id="1" Name="CVC" Type="Input Port" Detail="…">. . . </Component>
        <Component Id="2" Name="SMF" Type="Input Port" Detail="…">. . . </Component>
        <Component Id="3" Name="SAMF" Type="Input Port" Detail="…">. . . </Component>
        <Component Id="4" Name="LLD" Type="Input Port" Detail="…">. . . </Component>
        <Component Id="5" Name="ViChar" Type="Input Port" Detail="…">. . . </Component>
        <Component Id="5" Name="EDVC" Type="Input Port" Detail="…">. . . </Component>
</Library>
```

Figure 4.10 A library element defined in XML format.

4.4.2 Component

A component element defines a parameterizable entity in the router. Arbiter, Input Port, and Crossbar are three types of router components supported in HDPSO tool. Each router can have up to 5 input ports in the solution space. Each of these 5 input ports can be of any component of type ***'Input Port'*** and any configuration. A detailed component xml definition is shown in Figure 4.11.

```
<Component Id="0" Name="VC-Free" Type="Input Port" Detail="IP without any virtual channels.">
        <Configuration Id="0" Name="2-slots">
                <Properties>
                        <Property Name="# of Virtual Channels" Value="0" Unit="" />
                        <Property Name="Flit Size" Value="128" Unit="bits" />
                        <Property Name="Virtual Channel Depth" Value="256" Unit="bits" />
                </Properties>
                <Designs>
                        <Design Type="ASIC" Technology="32nm">
                                <Property Name="Supply_Voltage" Value="0.85" Unit="V" />
                                <Property Name="Frequency" Value="100" Unit="MHz" />
                                <Property Name="Area" Value="78590" Unit="µm^2" />
                                <Property Name="Static_Power" Value="1570" Unit="µW" />
                                <Property Name="Dynamic_Power" Value="65" Unit="µW" />
                        </Design>
                        <Design Type="FPGA" Device="Cyclone IV">
                                <Property Name="Logic_Elements" Value="3295" Unit="" />
                                <Property Name="Registers" Value="5795" Unit="bits" />
                        </Design>
                        <Design Type="ASIC" Technology="90nm"> . . .</Design>
                        <Design Type="ASIC" Technology="120nm"> . . .</Design>
                        <Design Type="ASIC" Technology="10nm"> . . .</Design>
                </Designs>
        </Configuration>
        <Configuration Id="1" Name="4-slots"> . . . </Configuration>
        <Configuration Id="2" Name="8-slots"> . . . </Configuration>
        <Configuration Id="3" Name="16-slots"> . . . </Configuration>
</Component>
```

Figure 4.11 A component element defined in XML format.

4.4.3 Configuration

A configuration is one combination of parameters in a parameterizable hardware module. For example, a hardware module contains two parameters ***FIFO_DEPTH*** and ***BUS_WIDTH***. If ***FIFO_DEPTH*** can be 1,4, or 8 and ***BUS_WIDTH*** can be 16, 32, or 64, then total configurations will be 9. This representation of configuration allows designers to automate synthesis of different configuration of modules and input into the optimization tool.

4.4.4 Design

A design represents all properties or synthesis results from one technology library. Designs can be of type ***'ASIC'*** or ***'FPGA'***. In case of ASIC, technology library size which was used to synthesize components should be passed. In case of FPGA, the component was synthesized using FPGA compilers and the device name for which it was synthesized should be passed. The tool is capable of optimizing NoC for FPGAs or ASICs. Reducing area and power in case of ASICs and reducing logic elements used in case of FPGAs.

4.4.5 Property

Property element is the basic element which stores a data value such as dynamic power, static power, registers, and frequency along with their units. When objects are created from parsing the input library xml, all property elements are copied over to the appropriate object properties.

4.5 A Sub-particle (Configuration Solution)

A sub-particle holds a single configuration solution named ***current_configuration***. It also holds ***current_cost*** which is sum of communication cost, static power, dynamic power, and area of current configuration solution multiplied by buffer penalty. Besides that, it also holds ***local_best_cost***, ***local_best_configuration***, ***mapping***, ***parent_particle_id***, static and dynamic power, area, and communication cost separately. The internal structure of the sub-particle is shown in Figure 4.12.

The array ***current_configuration*** holds universal IDs (uId) of component configurations. A uId is kept as an integer so that –ve values are permissible and components which are redundant can be represented by negative values. Allowed –ve uIds and what they represent is shown in Table 4.1. When a sub-particle is initialized, all uIds are set to undecided (ie. -4). Then the sub-swarm goes through configuration for topology stage (described in Section 4.6) where redundant components are marked and solutions are looked for only those components which are necessary.

Table 4.1 Negative Universal IDs (uIds) and their meanings.

uId	Meaning
0	NULL. Just initialized.
-1	This component is used but no configuration is mapped yet.
-2	This component is never used (due to application mapping).
-3	This component is not needed at all (due to topology).
-4	Undecided. It is not yet determined whether this component is need nor not.

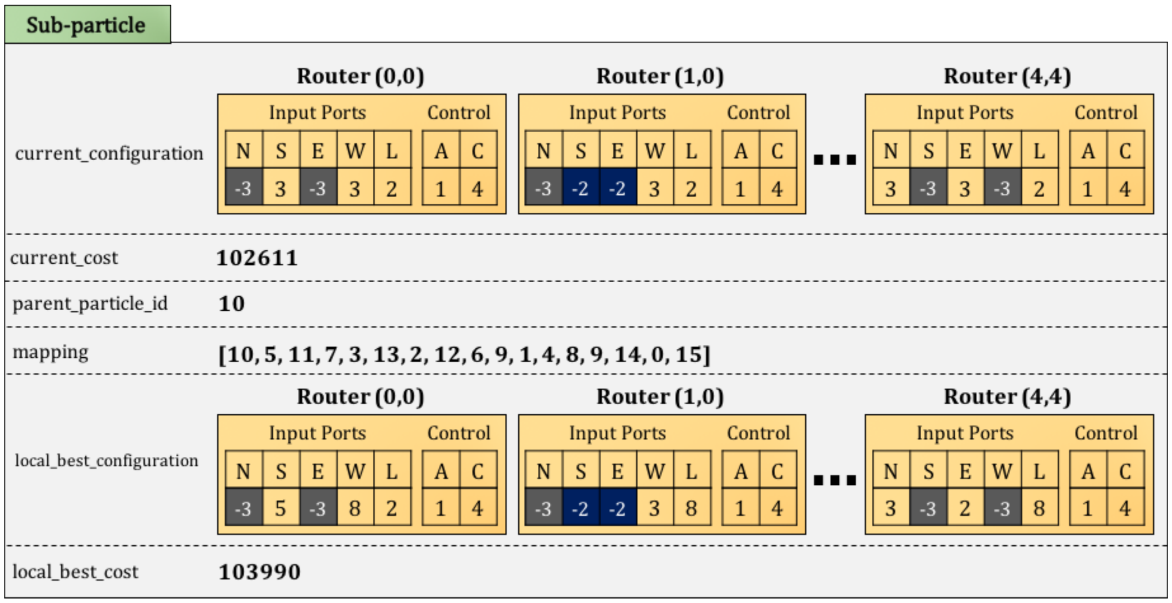


Figure 4.12 Internal structure of a sub-particle.

4.6 Configuration for Topology

When sub-particles (configuration solutions) are created based on the current parent particle (mapping solutions), the edge routers are identified and their outward input ports are marked as not needed as shown in Figure 4.13. It is to signal the swarm to ignore any optimal configuration for these input ports, e.g. east and north input port is not needed for the first router at the top left corner of the mesh. Then all the router input ports which participate in moving any traffic are marked. To accomplish this, each edge's source and destination location is identified and then packet path (XY routing) is followed while marking resources used on the way. After all the input ports are marked, the remaining input ports are marked as 'not needed' because no traffic edges pass through them. Configuration solutions for these input ports are not explored.

Finally, each router is checked whether it has any useful input port. If all the input ports of a router are marked as *'no edge passing'* then it is marked as *'dummy'* router and configuration for this router is not explored/optimized. It occurs when the number of cores is less than the number of tiles available in the topology. All such markings ensure that maximum time is spent on calculating optimal configuration for active and useful entities. Distances are calculated (as shown in Figure 4.14), and swap moves performed are also calculated ignoring the redundant entities, where the terms N, S, E, W, and L signify North, South, East, West, and Local input port configuration IDs respectively. While A and C stand for Arbiter and Crossbar configuration ID. The entire initialization algorithm pseudo-code is given in Algorithm 2.

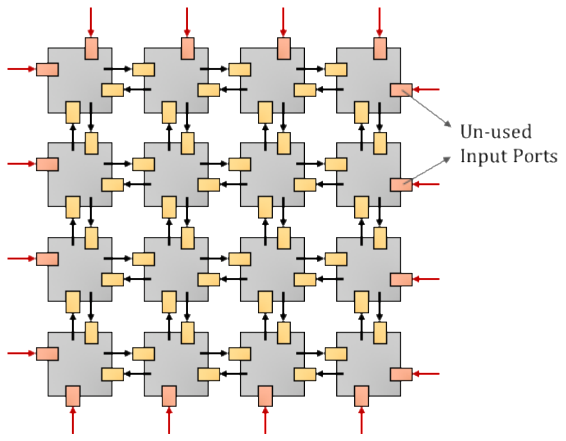
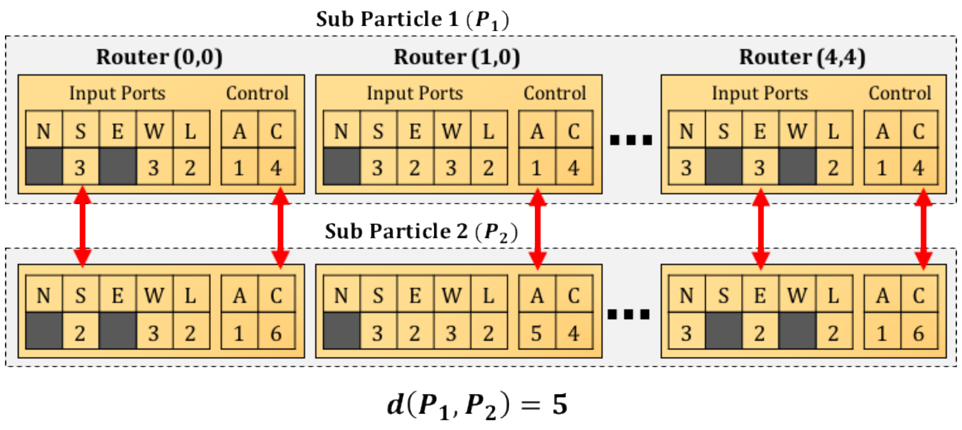


Figure 4.13 Identification of un-used input ports.

Algorithm 2: Sub-particle initialization

```
Input:  vector of sub-particle objects *sub_pars*
            vector of integer *mapping*
            vector of edges *coregraph*
        component library *lib*
Output: vector of initialized sub-particle objects *sub_pars*
// Iterate over all the sub-particles
for i = 0; i < sub_pars.size(); i++ do
    // Mark corner and edge routers to not have input ports which are not needed due to topology
    markCornerRouters(sub_pars, mapping, coregraph);
        // Mark input ports which do not have any traffic passing through
        markTrafficRoutes(sub_pars, mapping, coregraph);
        // Mark remaining routers as no edges are passing
        markNoTrafficRoutes(sub_pars, mapping, coregraph);
        // Mark dummy routers
        markDummyRouters(sub_pars, mapping, coregraph);
        // Configure routers with random configurations
        assignRandomConfigurations(sub_pars, mapping, coregraph, lib);
end
return sub_pars
```



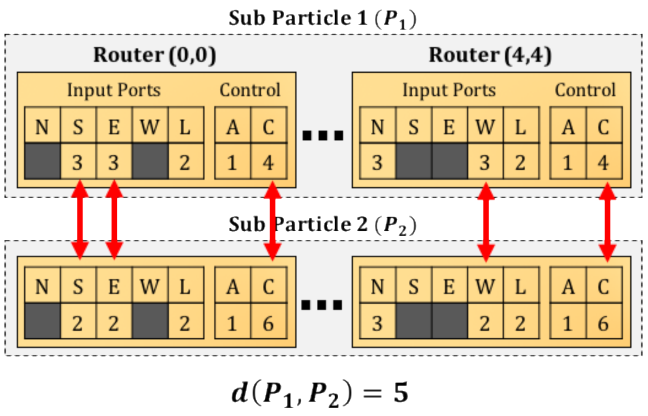


Figure 4.14 Determining distance between sub-particles (configuration solutions).

4.7 Initial Configuration Heuristics

Each main mapping particle is the possessor of a sub-swarm. At each iteration of the main particle, sub-swarm performs several iterations. The number of sub-swarm iterations per main swarm iteration is determined by the designer using trial-and-error and then set before running the algorithm. After construction of each main particle, sub-swarm is created and linked to the parent particle. After all the traffic routes are marked and useful entities are identified (as discussed in Section 4.3), random configuration IDs are selected from the available list of components. This process is depicted in Figure 4.15. This random initial heuristic placement does affect the final solution but ensures that particles are well-separated initially. Every component and its configuration is assigned a unique ID, which is the unique configuration of a component and its type. It is evident in Figure 4.14 that the ID 4 and 6 can only be assigned to a crossbar due to its type. Similarly, N, S, E, W and L can only contain IDs of type input port.

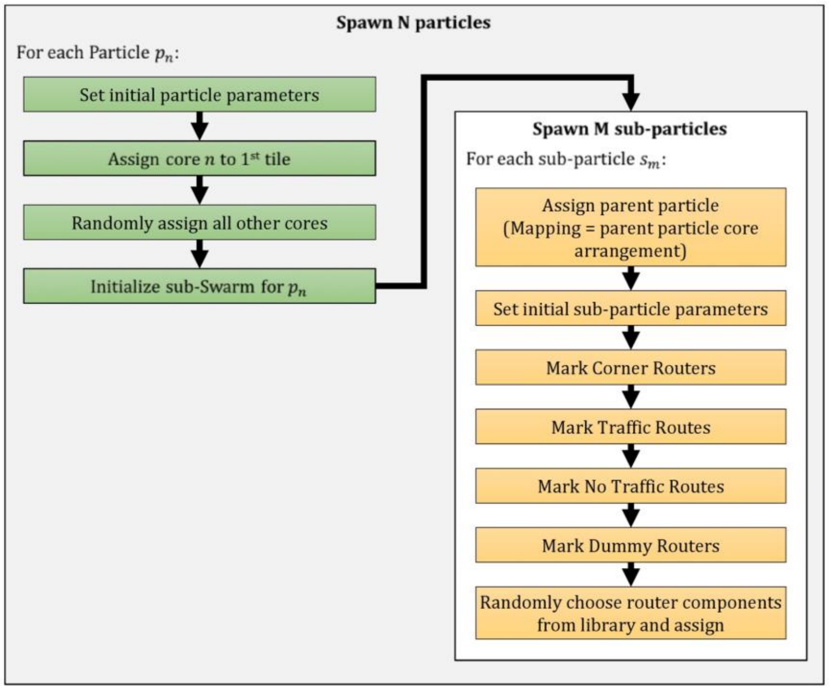

Spawn N particles
For each Particle p_n:
Set initial particle parameters
Assign core n to 1st tile
Randomly assign all other cores
Initialize sub-Swarm for p_n
Spawn M sub-particles
For each sub-particle s_m:
Assign parent particle (Mapping = parent particle core arrangement)
Set initial sub-particle parameters
Mark Corner Routers
Mark Traffic Routes
Mark No Traffic Routes
Mark Dummy Routers
Randomly choose router components from library and assign

Figure 4.15 Initial Configuration heuristics.

4.8 Sub-Swarm Movement

In each iteration of sub-particle, it may go through none or all 3 stages of optimization (ie. inertia, self-confidence, and swarm-confidence). In each stage, 1 move is applied to the particle which changes the mapping solution. This means that the sub particles need to start finding configuration solutions for this new mapping solution. Therefore, old sub-particles are destroyed, and new sub-particles are formed. The complete movement algorithm is depicted in Figure 4.16. A promising configuration that the sub-swarm might have found is kept in global memory of the main swarm. So, at the end of optimization if there are no other better solutions, the best global solution is given as optimal solution.

Tabu-ed memory is not kept and tabu-ed algorithm is not applied for sub-particles because there is no reflection mechanism and it incurs huge run time.

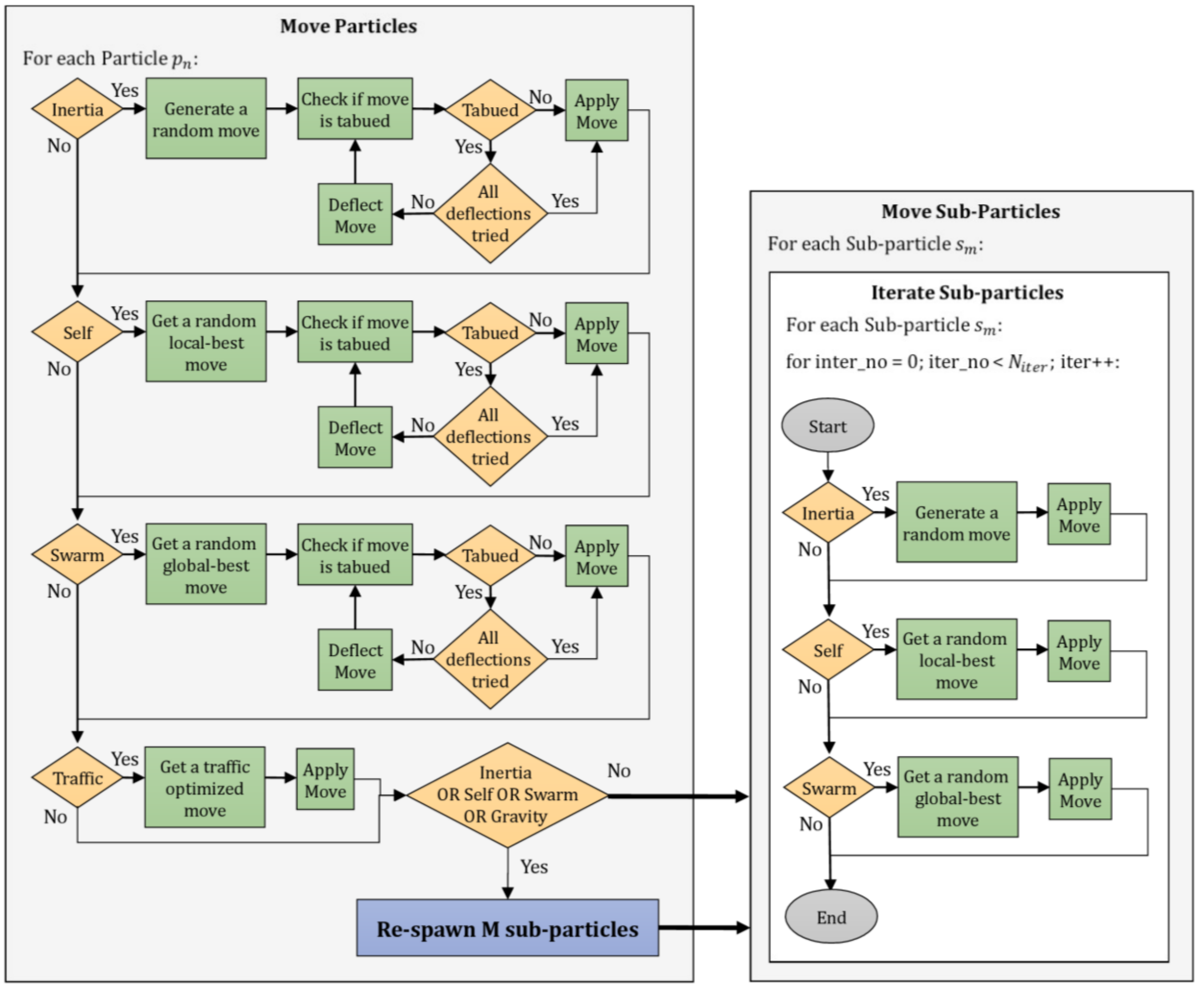


Figure 4.16 HDPSO sub-particle movement algorithm.

4.8.1 Mapping Edges to Links

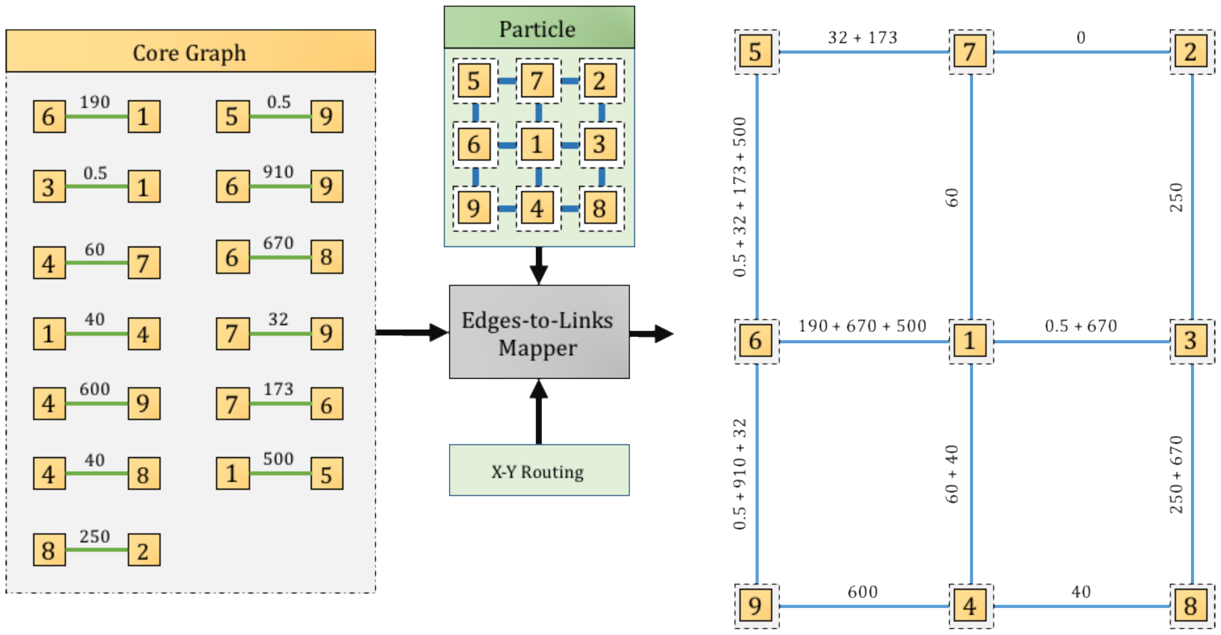


Figure 4.17 Link traffic volume calculation.

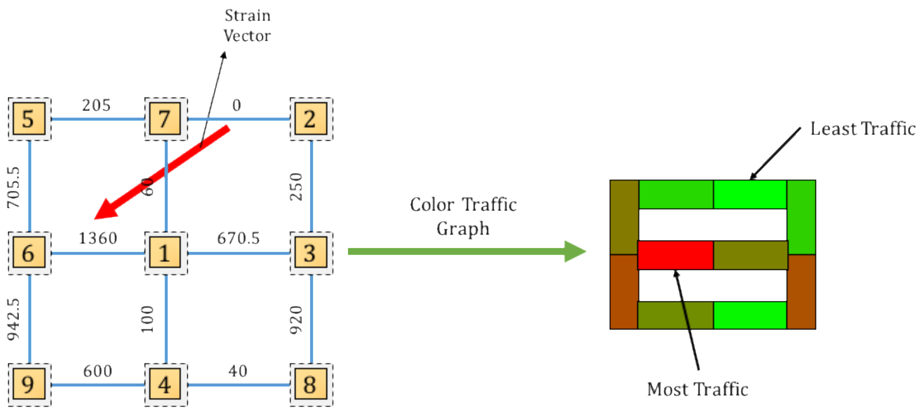


Figure 4.18 Link stress estimation.

4.8.2 Buffer Size Optimization

4.8.2.1 Conventional Virtual Channel (CVC)

In CVC architecture, a physical link supports several virtual channels that are multiplexed across the physical link as shown in Figure 4.19. Each virtual channel has its own buffer space, therefore there should be at least 2 buffers for each data flow. In our methodology, we have assumed virtual channels equal to the number of edges passing through the link. This ensures that each data path always gets assigned a virtual channel. Number of edges passing through a physical link is also known as data flows. Data flows can share a physical link.

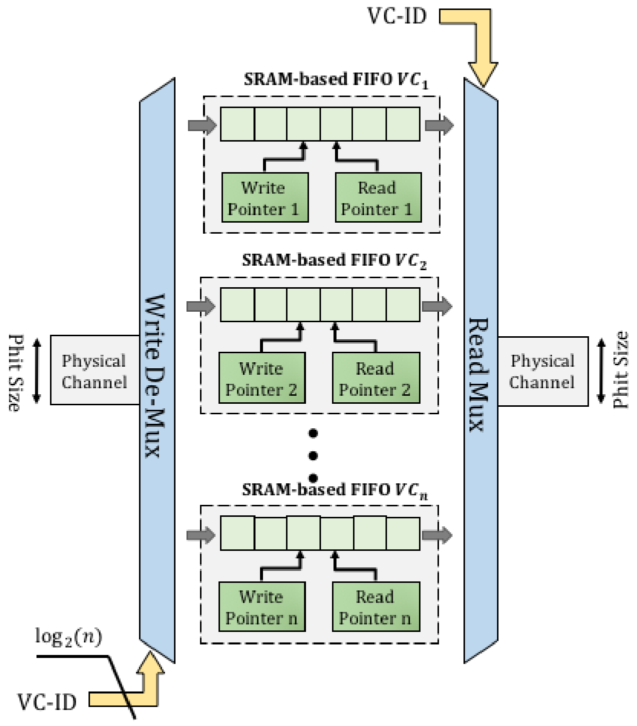


Figure 4.19 CVC input port architecture.

Since header flits allow the data flows to save their space in the arbitration queue, they should be allowed to send at least the header flit through router. This is the reason for allowing at least 1 buffer space to write 1 flit. However, in the same clock cycle, a flit should also be allowed to traverse out. We assume that the output ports are not buffered. This is the reason for 2nd buffer space. While the flit is being written into FIFO, another flit is being read for the output.

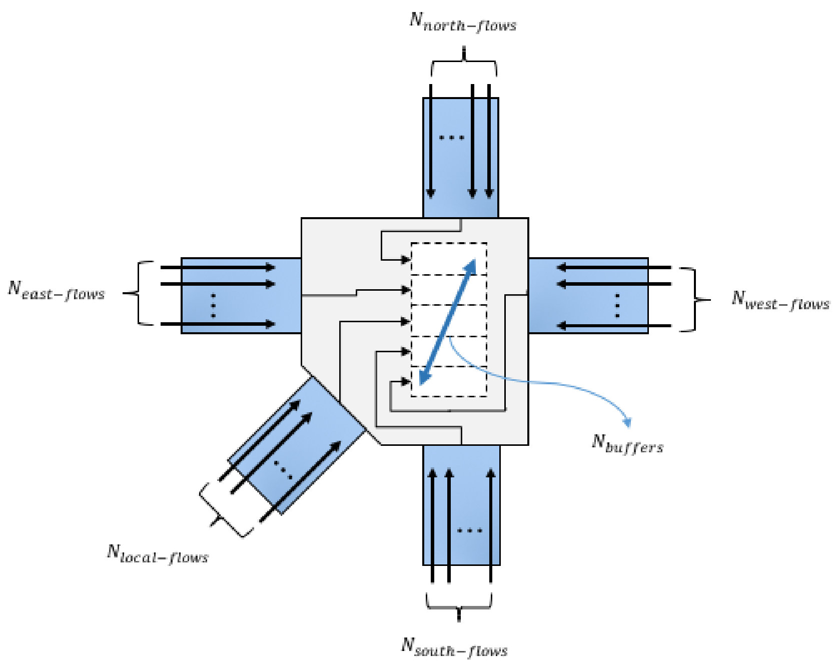


Figure 4.20 Determining minimum number of buffers required.

4.8.2.2 Dynamically Allocated Multi-Queues (DAMQ)

Coregraph provides information about how much (on-average) data travels over a period of 1 second, however it does not provide information on when does the data travel. For example, the data from one flow might travel within 1st 300 ms of a second while data from other flow traverses during 400ms to 800ms mark in a second. If the data traversal does not overlap, the buffering requirement is maximum of all flows. If the data traversal overlaps, the buffering requirement is sum of buffering requirement of both data flows. Time division of data flows can reduce the buffering requirements for architectures where VCs are dynamically allocated.

Figure 4.21 illustrates a 4-VC DAMQ input port where the addresses of flits are kept in a linked list table. The linked list table records the flit addresses per their *VC-ID* and in a First-Come-First-Serve (FCFS) basis.

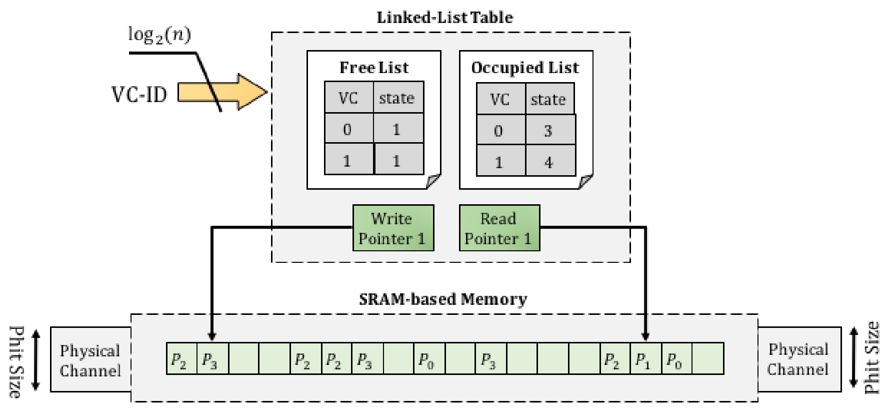


Figure 4.21 DAMQ VC input port architecture.

4.9 Convergence and Stopping Criteria

The design of the algorithm is such that if the main particle stays at the current mapping solution for a long period of time, more iterations are run for its sub-swarm that is looking for configuration solutions. However, when the main particle moves from one location to another, it carries with it its sub-swarm too and all the sub-particles explore the configurations for this new mapping. Commonly, the swarm particles of DPSO converge to an optimal point in search space near the end of optimization. It means that the particles have settled and optimization need not to continue. To detect this situation and stop the optimization, a normalized diversity factor is used that detects the average distance of the particles and sub-particles and stops the iterations if it is less that a certain threshold. The threshold we found empirically to work best was 5. This distance is calculated using the method shown in Figure 4.14. The number of configuration differences between the solutions is the distance between the sub-particles.

4.10 NoC Configuration Results

We have implemented our multi-swarm Hybrid DPSO based methodology (HDPSO) by using the formulation presented in this paper. The computer system for optimization has a quad-core Intel i7 CPU running at 2.7 GHz with 16GB RAM. The NoC links and the router components library used are based on 90nm technology. The area and power consumption values are determined by having a Verilog design of NoC routers and its components such as input-ports having buffer of 4 slots with 1-4 dynamic VCs, crossbar switch, VC allocators and arbiters. The channel link width is equal to the flit size of 16-bits and the overall router and target NoC architecture is based on EDVC methodology presented by Oveis-Gharan and Khan

[11]. The models which our tool uses is parametric and no simulation is performed while optimization. The EDVC router and its components' Verilog designs are synthesized by using the Synopsys design compiler.

4.10.1 Input Library

4.10.1.1 Crossbar Library

Table 4.2 Crossbar input library properties

Configuration	Flit Size	Designs	Property	Value
128-bit wide	128	ASIC (32nm)	Supply Voltage	0.85 V
			Frequency	100 MHz
			Area	6499 μm^2
			Static Power	121 μW
			Dynamic Power	6 μW
		FPGA (Cyclone IV)	Logic Elements	256
			Registers	0
16-bit wide	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	2502 μm^2
			Static Power	611 μW
			Dynamic Power	$- \mu W$
		FPGA (Stratix III)	Logic Elements	160
			Registers	0

4.10.1.2 Arbiter Library

Table 4.3 Arbiter input library properties

Configuration	Flit Size	Designs	Property	Value
2-slots #VC = 4 VC Depth = 2 flits	128	ASIC (32nm)	Supply Voltage	0.85 V
			Frequency	100 MHz
			Area	14274 μm^2
			Static Power	644 μW
			Dynamic Power	202 μW
		FPGA (Cyclone IV)	Logic Elements	2682
			Registers	280
4-slots #VC = 4 VC Depth = 4 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	28380 μm^2
			Static Power	1904 μW
			Dynamic Power	$- \mu W$
		FPGA (Stratix III)	Logic Elements	1116
			Registers	240

4.10.1.3 Input Port Library

Table 4.4 LLD input port library properties

Configuration	Flit Size	Designs	Property	Value
4-slots #VC = 4 VC Depth = 4 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	5809 μm^2
			Static Power	218 μW
			Dynamic Power	$- \mu W$
		FPGA (Stratix III)	Logic Elements	95
			Registers	112
8-slots #VC = 4 VC Depth = 8 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	10328 μm^2
			Static Power	370 μW
			Dynamic Power	$- \mu W$
		FPGA (Stratix III)	Logic Elements	180
			Registers	204
16-slots #VC = 4 VC Depth = 16 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	19813 μm^2
			Static Power	688 μW
			Dynamic Power	$- \mu W$
		FPGA (Stratix III)	Logic Elements	332
			Registers	388
32-slots #VC = 4 VC Depth = 32 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	39147 μm^2
			Static Power	1306 μW
			Dynamic Power	$- \mu W$
		FPGA (Stratix III)	Logic Elements	670
			Registers	764

Table 4.5 ViChaR input port library properties

Configuration	Flit Size	Designs	Property	Value
4-slots #VC = 4 VC Depth = 4 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	6646 μm^2
			Static Power	319 μW
			Dynamic Power	$-\ \mu W$
		FPGA (Stratix III)	Logic Elements	75
			Registers	132
8-slots #VC = 4 VC Depth = 8 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	21274 μm^2
			Static Power	1236 μW
			Dynamic Power	$-\ \mu W$
		FPGA (Stratix III)	Logic Elements	306
			Registers	392
16-slots #VC = 4 VC Depth = 16 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	48463 μm^2
			Static Power	2968 μW
			Dynamic Power	$-\ \mu W$
		FPGA (Stratix III)	Logic Elements	548
			Registers	896
32-slots #VC = 4 VC Depth = 32 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	109849 μm^2
			Static Power	6989 μW
			Dynamic Power	$-\ \mu W$
		FPGA (Stratix III)	Logic Elements	1183
			Registers	2040

Table 4.6 EDVC-FRW input port library properties

Configuration	**Flit Size**	**Designs**	**Property**	**Value**
4-slots #VC = 4 VC Depth = 4 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	4991 μm^2
			Static Power	106 μW
			Dynamic Power	$-\ \mu W$
		FPGA (Stratix III)	Logic Elements	83
			Registers	107
8-slots #VC = 4 VC Depth = 8 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	9524 μm^2
			Static Power	150 μW
			Dynamic Power	$-\ \mu W$
		FPGA (Stratix III)	Logic Elements	206
			Registers	186
16-slots #VC = 4 VC Depth = 16 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	19448 μm^2
			Static Power	240 μW
			Dynamic Power	$-\ \mu W$
		FPGA (Stratix III)	Logic Elements	441
			Registers	340
32-slots #VC = 4 VC Depth = 32 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	40716 μm^2
			Static Power	413 μW
			Dynamic Power	$-\ \mu W$
		FPGA (Stratix III)	Logic Elements	964
			Registers	646

Table 4.7 EDVC-FW input port library properties

Configuration	Flit Size	Designs	Property	Value
4-slots #VC = 4 VC Depth = 4 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	4674 μm^2
			Static Power	108 μW
			Dynamic Power	$- \mu W$
		FPGA (Stratix III)	Logic Elements	78
			Registers	97
8-slots #VC = 4 VC Depth = 8 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	8687 μm^2
			Static Power	162 μW
			Dynamic Power	$- \mu W$
		FPGA (Stratix III)	Logic Elements	174
			Registers	174
16-slots #VC = 4 VC Depth = 16 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	17016 μm^2
			Static Power	263 μW
			Dynamic Power	$- \mu W$
		FPGA (Stratix III)	Logic Elements	324
			Registers	327
32-slots #VC = 4 VC Depth = 32 flits	16	ASIC (90 nm)	Supply Voltage	1.2 V
			Frequency	200 MHz
			Area	34295 μm^2
			Static Power	457 μW
			Dynamic Power	$- \mu W$
		FPGA (Stratix III)	Logic Elements	727
			Registers	632

Table 4.8 shows the results for some well-known benchmark core-graphs for comparison purposes. Picture in Picture (PIP), MPEG-4 Decoder (MPEG4), and Video Object Plane Decoder (VOPD) are some of the typical core graphs used earlier to evaluate NoC mapping techniques. We are also presenting the mapping of these core-graphs to suitable size 2D NoCs in Fig. 5. The PIP, MPEG4, VOPD and DVOPD core graphs are given in the survey paper by Sahu et al. [26]. The cost for our HDPSO based optimized NoC mapping and configuration solution is compared with non-optimized solutions for the NoC mapping of same core-graphs. In non-optimized method, all the routers used have four input ports with four VCs per input port slots, where for our HDPSO optimized solutions, routers have different number of virtual channels and input ports depending on the number of edges to each core and the volume of traffic among different cores.

Table 4.8 Area/Power for PIP and MPEG4 in optimal, generous, and minimal configurations

Name	Cores	Mesh Size	Particle Cost	Method	#Buffers/VC	Input Port Architecture	NoC Area (μm^2)	NoC Power (μW)	Buffer Penalty	Total Configuration Cost
PIP	8	3 x 3	640	Generous	8	LLD	376152	24824	1	401616
						ViChar	470416	32560		503616
						EDVC-FRW	363176	22168		385984
						EDVC-FW	353944	22280		376864
				HDPSO	Varying	LLD	344519	23760	1	368919
						ViChar	368020	26141		394801
						EDVC-FRW	331445	21860		353945
						EDVC-FW	325853	21902		348395
				Minimal	4	LLD	340000	23608	1.04	378817
						ViChar	470416	32560	1	503616
						EDVC-FRW	363176	22168		385984
						EDVC-FW	353944	22280		376864
MPEG4	12	4 x 4	3567	Generous	8	LLD	574556	37606	1	615729
						ViChar	726898	50076		780541
						EDVC-FRW	554288	33402		591257
						EDVC-FW	539603	33582		576752
				HDPSO	Varying	LLD	533885	36238	1	573690
						ViChar	595246	41823		640636
						EDVC-FRW	513491	33006		550064
						EDVC-FW	503486	33096		540149
				Minimal	4	LLD	533885	36238	1.06	608111
						ViChar	595246	41823	1	640636
						EDVC-FRW	513491	33006		550064
						EDVC-FW	503486	33096		540149
VOPD	16	4 x 4	4425	Generous	8	LLD	834928	52608	1	891961
						ViChar	1111024	75008		1190457
						EDVC-FRW	802544	45536		852505
						EDVC-FW	777384	45856		827665
				HDPSO	Varying	LLD	795598	51282	1	851305
						ViChar	891604	61253		957282
						EDVC-FRW	734549	44876		783850
						EDVC-FW	717189	45046		766660
				Minimal	4	LLD	834928	52608	1.08	963318
						ViChar	1111024	75008	1.02	1214266
						EDVC-FRW	802544	45536	1.02	869555
						EDVC-FW	777384	45856	1.02	844218
DVOPD	26	6 x 6	22962	Generous	8	LLD	1248048	67408	1	1338418
						ViChar	1961984	124448		2109394
						EDVC-FRW	1183504	51536		1258002
						EDVC-FW	1124864	52336		1200162
				HDPSO	Varying	LLD	1238912	67080	1	1328954
						ViChar	1372730	87564		1483256
						EDVC-FRW	1003900	49780		1076642
						EDVC-FW	964950	50162		1038074
				Minimal	4	LLD	958832	57680	1.06	1101842
						ViChar	1025792	65760	1	1114514
						EDVC-FRW	893392	48720		965074
						EDVC-FW	868032	48880		939874

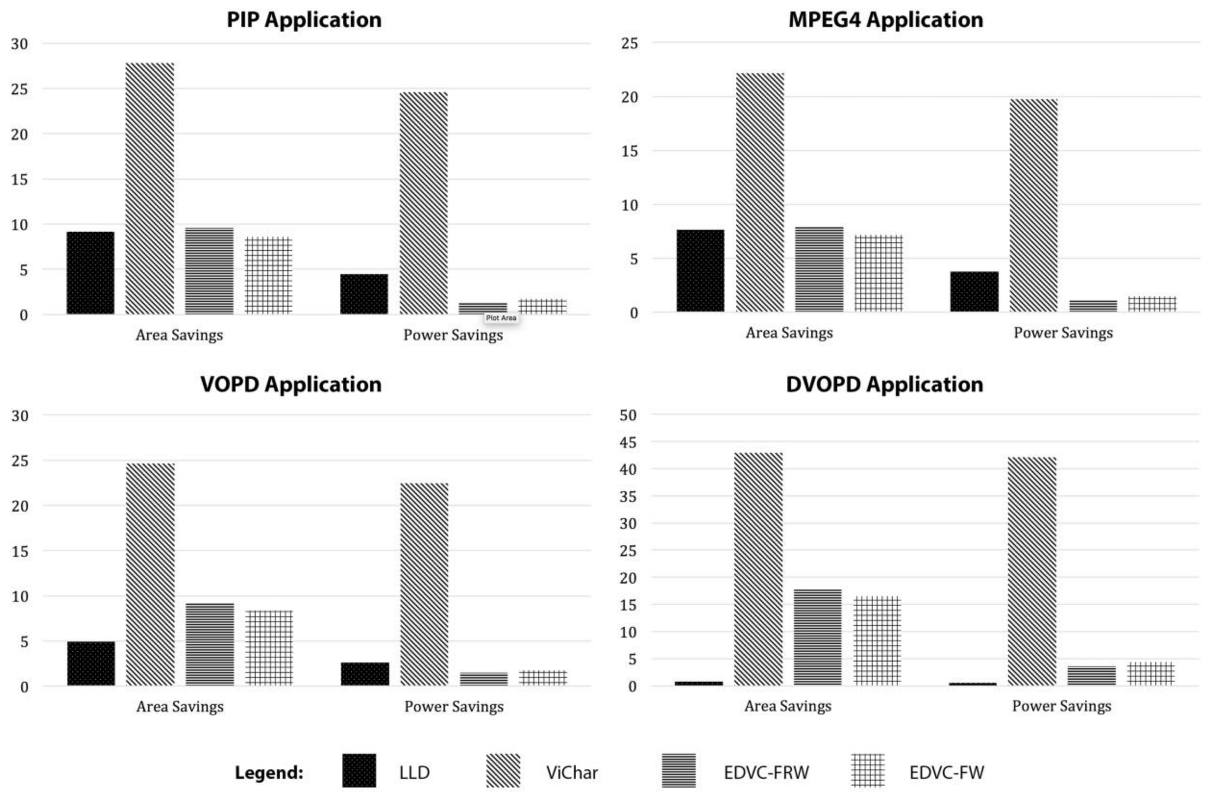
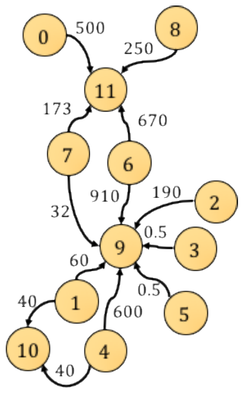
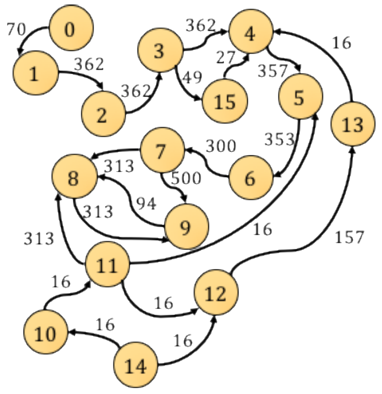
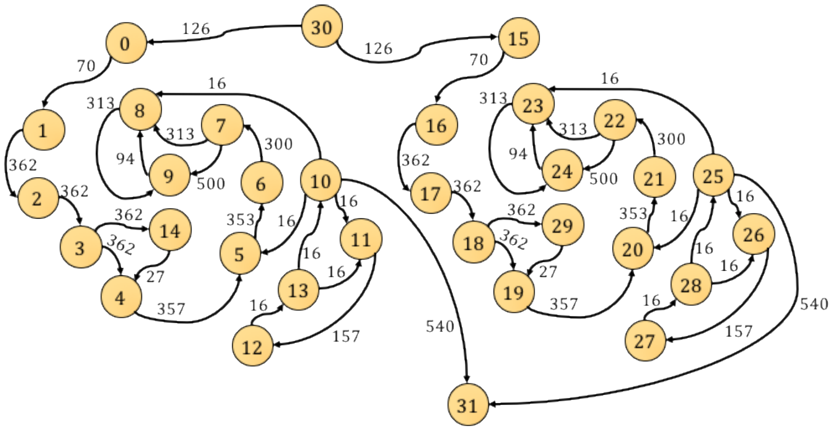


Figure 4.22 Power and area % savings observed when comparing HDPSO optimized configuration to un-optimized generous configuration.

It can be seen from the input libraries of input port that the increase in power is not steep as compared to increase in area when more buffer slots are added to the input port. This is one of the reasons why power savings are relatively small than area savings in Figure 4.22. Overall, ViChar input port saw the most improvement. This was because higher number of VC slots in ViChar resulted in almost exponential increase in input port area and power. On average, power savings for real world applications ranged from 1 - 43% and area savings ranged from 22 – 44 %.


(a)


(b)


(c)

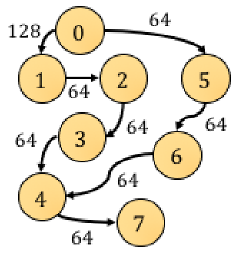

(d)

Figure 4.23 NoC coregraphs (a) MPEG4 (b) 16-core VOPD (c) DVOPD (d) PIP.

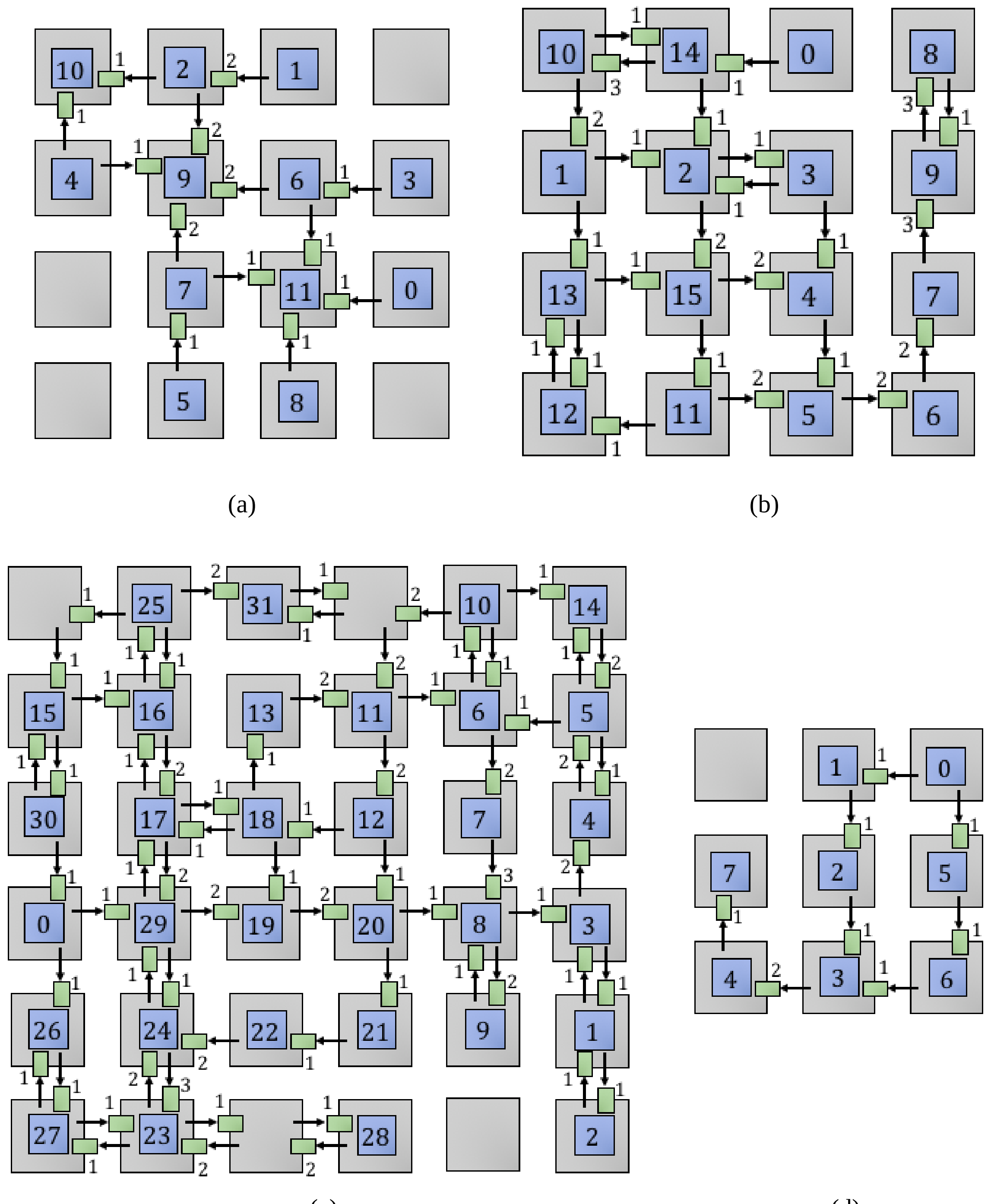


Figure 4.24 NoC configuration and mapping solutions (a) MPEG4 (b) VOPD (c) DVOPD (d) PIP.

In order to give equal weightage to area, power, and communication cost, $\alpha, \beta, and\ \gamma$ were all set to 1. It is observed that the applications with lower edges-to-cores ratio (e.g. MPEG4) have better cost improvement due to configuration. This is also shown in Figure 4.25. This was due to the fact that more of the input ports could be eliminated if there are less edges in the application. The improvement with regards to the chip-area is in the range of 30-120%, and power improvement varies from 8-40%. Overall, the total cost improvement is in the range of 20-110%. The optimized NoC mappings of MPEG4, VOPD, DVOPD and PIP are presented in Figure 4.24 and these are based on NoC configuration and mapping achieved by following our HDPSO methodology. The green or small rectangular boxes signify the input ports and the number beside it represents the number of virtual channels needed for efficient implementation. Due to the use of smaller and simpler application core graphs, the VCs required at any of the input port is less than a maximum of four VCs. The NoC chip area and power are saved by using less number of input ports and fewer virtual channels.

After each iteration of mapping optimization, several iterations of configuration optimization are run in order to try different configurations on the mapping solution. The combined results from two swarms are used to determine next position of main (mapping) particles and sub-particles (configuration). The solution with the best overall cost is selected and given as optimal solution after the swarm has converged onto a relatively small space (determined by the swarm divergence factor). It was observed that the applications with lower edges-to-cores ratio had better cost improvement and the solution had a look of mesh topology but in reality was a custom-mesh topology (See Figure 2.1g). This was due to uneven distribution of input ports and router. We intend to expand our library of pre-synthesized routers and include optimization for virtual channel buffer depth.

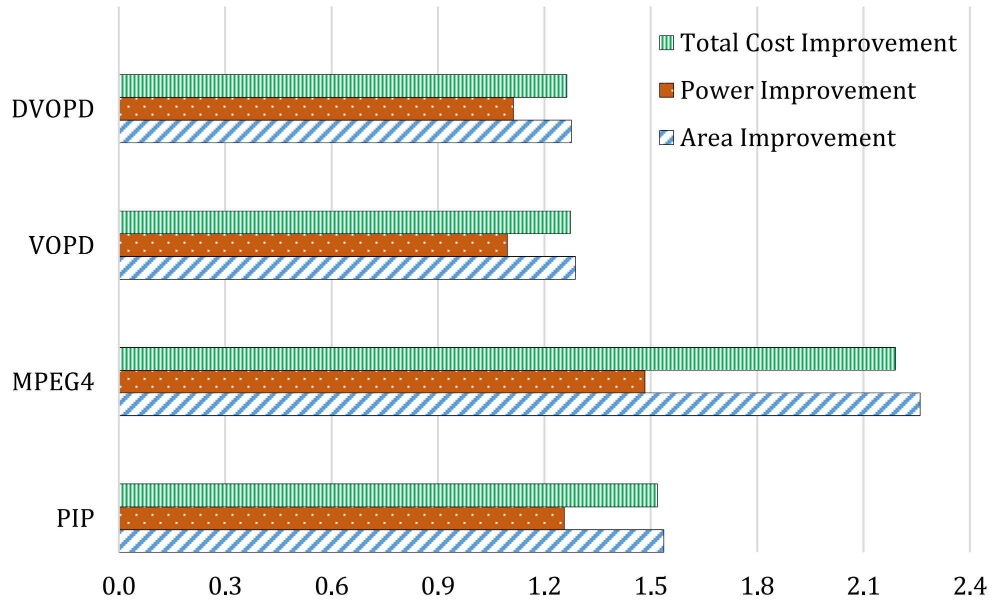


Figure 4.25 Cost improvements due to configuration optimization.

4.11 Summary

After each iteration of mapping optimization, several iterations of configuration optimization are run in order to try different configurations on the mapping solution. The combined results from two swarms are used to determine next position of main (mapping) particles and sub-particles (configuration). The solution with the best overall cost is selected and given as optimal solution after the swarm has converged onto a relatively small space (determined by the swarm divergence factor). It was observed that the applications with lower edges-to-cores ratio had better cost improvement and the solution had a look of mesh topology but in reality was a custom topology. This was due to uneven distribution of input ports and router. We also performed optimization for virtual channel buffer depth during the configuration optimization. A push-pull force was established between average duty factor of NoC links and buffer depth in order to keep the performance of the network high while reducing NoC power and area.

Chapter 5

Conclusion

A Hybrid Swarm Optimization method is presented for NoC mapping, which increases the efficiency of traditional discrete particle swarm optimization based mapping. It has a Tabu-ing of search points that swarm particles have visited. Since the search space was of combinatorial discrete nature, deflection technique is used to encourage the swarm particles to explore much more of the solution space before converging to the optimal solution. If the search space is of continuous nature, then other methods such as Bezier curve would have been employed to force the particle to curve around the Tabu-ed area.

If a swap move results in a solution which is already tried (i.e. Tabu-ed), the proposed moves are deflected in the other directions until there is no other way to allow the original move. This causes the algorithm to explore more of the search space before converging onto the global best solution. The HDPSO based NoC mapping is employed for various popular applications PIP, MPEG-4, VOPD, and DVOPD as well as large synthetic traffic patterns and our HDPSO methodology performs better than other comparable NoC mapping algorithms.

Standard DPSO based method has also been implemented for NoC mapping and quality of the results is compared with our HDPSO based mapping. It was found that HDPSO performs 15% to 216% better than traditional DPSO technique. Our tool performed better in particular for traffic patterns which contain ring communication structure such as Neighbor, Tornado, and Shuffle traffic patterns. It is also concluded that remembering the past solutions and proposing new solutions by our HDPSO algorithm enables it to produce better mapping solutions.

A new Hybrid Multi-Swarm Optimization (HDPSO) method is presented for NoC mapping and configuration, which increases the efficiency of traditional discrete particle swarm optimization and uses a library of pre-synthesized router components. Since there are two optimization problems, multi-swarm method was used to explore each solution space. Due to combinatorial discrete nature of the search space, deflection technique is used to encourage the swarm particles to explore much more of the solution space before converging to the optimal solution.

References

[1] T. Bjerregaard and S. Mahadevan, "A Survey of Research and Practices of Network-on-chip," *ACM Comput. Surveys,* vol. 38, no. 1, pp. 1-51, June 2006.

[2] M. Oveis-Gharan and G. N. Khan, "Efficient Dynamic Virtual Channel Organization and Architecture for NoC Systems," *IEEE Trans. on VLSI Syst.*, vol. 24, no. 2, pp. 465-478, February 2016.

[3] M. Oveis-Gharan and G. N. Khan, "Dynamic virtual channel and index-based arbitration based Network on Chip router architecture," in *Proc. Inter. Conf. on High Perf. Comput. & Sim. (HPCS),* Innsbruck, Austria, 2016, pp. 96-103.

[4] M. Oveis-Gharan and G. N. Khan, "Adaptive VC Organization and Arbitration for Efficient NoC Design," in *Proc. 10th Int. Symp. on Emb. Multicore/Many-core SoCs (MCSoC),* Lyon, France, 2016, pp 31-38.

[5] M. Oveis-Gharan and G. N. Khan, "Packet-based Adaptive Virtual Channel Configuration for NoC Systems," *Procedia Comput. Sc.*, vol. 34, no. Complete, pp. 552-558, January 2014.

[6] M. Oveis-Gharan and G. N. Khan, "Statically adaptive multi FIFO buffer architecture for network on chip," *Microprocessors and Microsystems,* vol. 39, no. 1, pp. 11-26, 2015.

[7] M. Oveis-Gharan and G. N. Khan, "A Novel Virtual Channel Implementation Technique for Multi-core On-chip Communication," in *Proc. 3rd Workshop on App. for Multi-Core Arch. (WAMCA),* New York, USA, 2012, pp. 36-41.

[8] J. Kennedy and R. Eberhart, "Particle Swarm Optimization," in *Proc. of IEEE Int. Conf. on Neural Networks*, Perth, Australia, 1995, pp 1942-1948.

[9] F. Glover, "Tabu Search," *Journal on Comput.,* no. 1, pp. 190-206, 1989.

[10] X. Chen and L.-S. Peh, "Leakage Power Modeling and Optimization in Interconnection Networks," in *Proc. Int. Symp. on Low Power Elec. and Design,* Seoul, South Korea, 2003, pp. 90-95.

[11] W. Dally and B. P. Towles, "Buffered Flow Control," in *Principles and Practices of Interconnection Networks*, 1st ed., San Francisco, CA, USA: Morgan Kaufmann, 2003, pp. 233-256.

[12] S. P. Singh, S. Bhoj, D. Balasubramanian, T. Nagda, D. Bhatia and P. Balsara, "Network interface for NoC based architectures," *Int. Journal of Electronics*, vol. 94, no. 5, pp. 531-547, May 2007.

[13] W. Dally and B. P. Towles, "Adaptive Routing," in *Principles and Practices of Interconnection Networks*, 1st ed., San Francisco, CA, USA: Morgan Kaufmann, 2003, pp. 189-196.

[14] E.-J. Chang, H.-K. Hsin, C.-H. Chao, S.-Y. Li and A.-Y. Wu, "Regional ACO-Based Cascaded Adaptive Routing for Traffic Balancing in Mesh-Based Network-on-Chip Systems," *IEEE Trans. Comput.*, vol. 64, no. 3, pp. 868-875, March 2015.

[15] S. Liu, T. Chen, L. Li, X. Li, M. Zhang, C. Wang, H. Meng, X. Zhou and Y. Chen, "FreeRider: Non-Local Adaptive Network-on-Chip Routing with Packet-Carried Propagation of Congestion Information," *IEEE Trans. on Parallel and Distributed Sys.*, vol. 26, no. 8, pp. 2272-2285, August 2015.

[16] J. Kim, C. Nicopoulos and D. Park, "A gracefully degrading and energy-efficient modular router architecture for on-chip networks," in *Proc. 33rd International Symposium on Computer Architecture*, Boston, MA, USA, 2006, pp. 4-15.

[17] M. Oveis-Gharan and G. N. Khan, "Index-based round-robin arbiter for NoC routers," in *Proc. IEEE CS Symp. on VLSI*, Montpellier, France, 2015, pp. 62-67.

[18] A. Jalabert, S. Murali, L. Benini and G. D. Micheli, "Xpipes Compiler: a tool for instantiating application specific networks on chip," in *Proc. of DATE Conf.*, Paris, France, 2004, pp. 884-889.

[19] G. Leary, K. Mehta and K. S. Chatha, "Performance and resource optimization of NoC router architecture for master and slave IP cores," in *Proc. of Int. Conf. on Hardware/Software Codesign and Sys. Synthesis (CODES+ISSS)*, Salzburg, Austria, 2007, pp. 155-160.

[20] A. A. Morgan, H. Elmiligi, M. W. El-Kharashi and F. Gebali, "Unified multi-objective mapping and architecture customization of networks-on-chip," *IET Computer & Digital Techniques*, vol. 7, no. 6, pp. 282-293, 2013.

[21] J. Hu and R. Marculescu, "Energy-aware mapping for tile-based NoC architectures under performance constraints," in *Proc. ASP-DAC*, Kitakyushu, Japan, 2003, pp. 233-239.

[22] G. Jiang, Z. Li, F. Wang and S. Wei, "Mapping of Embedded Applications on Hybrid Networks-on-Chip with Multiple Switching Mechanisms," IEEE *Embedded Sys. Letters,* vol. 7, no. 2, pp. 59-62, June 2015.

[23] S. Murali and G. De Micheli, "Bandwidth-constrained mapping of cores onto NoC architectures," *Proc. DATE Conf. and Exhibition,* February 2004, pp. 896-901.

[24] P. Sahu, T. Shah, K. Manna and S. Chattopadhyay, "Application Mapping Onto Mesh-Based Network-on-Chip Using Discrete Particle Swarm Optimization," *IEEE Trans. on VLSI Sys.*, vol. 22, no. 2, pp. 300-312, February 2014.

[25] A. Bender, "MILP based task mapping for heterogeneous multiprocessor systems," in *Proc. of EURO-DAC*, Geneva, Switzerland, 1996, pp. 190-197.

[26] P. K. Sahu and S. Chattopadhyay, "A survey on Application Mapping Strategies for Network-on-Chip design," *Journal of Sys. Arch.,* vol. 59, no. 1, pp. 60-76, January 2013.

[27] B. Yang, L. Guang, T. Santti and J. Plosila, "T(K)-SA: Accelerated Simulated Annealing Algorithm for Application Mapping on Networks-on-chip," in *Proc. 14th Annual Conf. on Genetic and Evolutionary Comp.*, New York, NY, USA, 2012, pp. 1191-1198.

[28] A. Tino and G. Khan, "Power and Performance Tabu Search Based Multicore Network-on-Chip Design," in *Proc. 39th Int. Conf. on Parallel Proc. Workshops (ICPPW)*, 2010, pp. 74-81.

[29] N. Nedjah, M. V. C. de Silva and L. de M. Mourelle, "Customized computer-aided application mapping on NoC infrastructure using multi-objective optimization," *Journal of Sys. Arch.,* vol. 57, pp. 79-94, 2011.

[30] N. Nedjah, M. V. C. de Silva and L. de M. Mourelle, "Preference-based multi-objective evolutionary algorithms for power-aware application mapping on NoC platforms," *Expert Sys. with App.,* vol. 39, pp. 2771-2782, 2012.

[31] Z. Li, Y. Liu and M. Cheng, "Solving NoC mapping problem with improved particle swarm algorithm," in *Proc. of Int. Conf. on Advanced Comp. Intelligence (ICACI)*, Hangzhou, China, 2013, pp. 12-16.

[32] S. Mirjalili and S. Hashim, "A new hybrid PSOGSA algorithm for function optimization," in *Proc. Int. Conf. on Comp. and Info. App. (ICCIA)*, 2010, pp. 374-377.

[33] Y. Z. Tei, M. N. Marsono, N. Shaikh-Husin and Y. W. Hau, "Performance and resource optimization of NoC router architecture for master and slave IP cores," in *Proc. of IEEE Int. Symp. Circuits and Systems*, Beijing, China, 2013, pp. 1228-1231.

[34] K. Goossens, J. Dielissen, O. P. Gangwal, S. G. Pestana, A. Radulescu and E. Rijpkema, "A design flow for application-specific networks on chips with guaranteed performance to accelerate SoC design and verification," in *Proc. of DATE Conference*, Munich, Germany, 2005, pp. 1182-1187.

[35] L. Benini and G. D. Micheli, “Tools for NoC Design Analysis and Synthesis of NoCs Present tools (Bones, Xpipes) and future outlook”, *Networks on Chips*, San Francisco, CA, USA: Morgan Kaufmann, 2006.

[36] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi and G. D. Micheli, "Xpipes Lite: a synthesis oriented design library for networks on chips," in *Proc. of DATE Conference*, Munich, Germany, 2005, pp. 1188-1193.

[37] A. T. Tran and B. M. Baas, "NoCTweak: a Highly Parameterizable Simulator for Early Exploration of Performance and Energy of Networks On-Chip," in *Technical Report of VLSI Computation Lab, ECE Department*, Davis, CA, USA, 2012, pp. 1-12.

[38] K.-l. Tsai, H.-t. Chen and Y.-A. Lin, "Power and Area Efficiency NoC Router Design for Application-Specific SoC by Using Buffer Merging and Resource Sharing," *Tans. on Design Automation of Elec. Sys.*, vol. 19, no. 4, pp. Art. 36, Aug. 2014.

[39] D. R. C. Silva and C. J. A. Bastos-Filho, "A Multi-Objective Particle Swarm Optimizer Based on Diversity," in *Proc. 2nd Int. Conf. on Intel. Sys. and App.* , Venice, Italy, 2013, pp. 109-114.

[40] J. Hu and R. Marculescu, "Energy- and performance-aware mapping for regular NoC architectures," *Trans. on CAD of Integ. Circuits and Syst.*, vol. 24, no. 4, pp. 551-562, April 2005.

[41] P. K. Sahu, N. Shah, K. Manna and S. Chattopadhyay, "A new application mapping algorithm for mesh based Network-on-Chip design," in *Proc. IEEE India Conf. (INDICON)*, Kolkata, India, 2010, pp. 1-4.

[42] P. K. Sahu, P. Venkatesh, S. Gollapalli and S. Chattopadhyay, "Application mapping onto Mesh Structured Network-on-Chip Using Particle Swarm Optimization," in *Proc. IEEE Comp. Soc. Symp. on VLSI (ISVLSI)*, Chennai, India, 2011, pp. 335-336.

[43] R. P. Dick, D. L. Rhodes and W. Wolf, "TGFF: Task Graphs For Free," in *Proc. Int. Workshop Hardware/Soft. Codesign (CODES)*, Washington, USA, 1998, pp. 97-101.

[44] W. Dally and B. P. Towles, "Closing the Loop with Credits," in *Principles and Practices of Interconnection Networks*, 1st ed., San Francisco, CA, USA: Morgan Kaufmann, 2003, pp. 233-256.

[45] V. Palaniveloo, J. Ambrose and A. Sowmya, "Improving GA-Based NoC Mapping Algorithms Using a Formal Model," *Proc. IEEE Comp. Soc. Ann. Symp. on VLSI (ISVLSI)*, 2014, pp. 344-349.

[46] T. Bjerregaard and S. Mahadevan, "A Survey of Research and Practices of Network-on-chip," *ACM Comput. Surv.*, vol. 38, no. 1, pp. 1-51, June 2006.

[47] Y. Liu, Y. Ruan, Z. Lai and W. Jing, "Energy and thermal aware mapping for mesh-based NoC architectures using multi-objective ant colony algorithm," *Proc. 3rd Int. Conf. on Comp. Research and Dev. (ICCRD)*, vol. 3, pp. 407-411, 11-13 March 2011.

[48] F. Samman, T. Hollstein and M. Glesner, "Pipeline control-path effects on area and performance of a wormhole-switched network-on-chip," *Intl. Jour. of Electrical, Computer, Energetic, Electronic, and Communication Engineering*, vol. 4, pp. 951-959, 2009.

[49] H. T. Kung and K. Chang, "Receiver-oriented adaptive buffer allocation in credit-based flow control for ATM networks," in *Proc. Comp. and Comm. Soc. Bringing Info. to People*, Boston, MA, USA, 1995, pp. 239-252.

Glossary

2D	2 Dimensional
3D	3 Dimensional
ACG	Application Core Graph
AM	Application Mode
AV	Audio-Video
CG	Core Graph
CGG	Core Graph Generator
CPN	Congestion Propagation Network
DPSO	Discrete Particle Swarm Optimization
DVOPD	Dual Video Object Plane Decoder
GA	Genetic Algortihm
HDPSO	Hybrid Discrete Particle Swarm Optimization
IP	Intellectual Property
MPEG	Moving Picture Experts Group
MWD	Multi-Window Display
NoC	Network-on-Chip
PSO	Particle Swarm Optimization
PIP	Picture-in-Picture

SA	Simulated Annealing
SIP	Silicon Intellectual Property
SoC	System-on-Chip
TGFF	Task Graphs For Free
VOPD	Video Object Plane Decoder