Hierarchical Agent-based Adaptation for Self-Aware Embedded Computing Systems

Liang Guang

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in Auditorium Beta on December 10th, 2012, at 12 noon.

University of Turku
Department of Information Technology
20014 Turun Yliopisto

2012

Supervisors

Professor Hannu Tenhunen
Professor Jouni Isoaho
Associate Professor Juha Plosila
Associate Professor Pasi Liljeberg
Department of Information Technology
University of Turku
Turku, Finland

Reviewers

Associate Professor Waltenegus Dargie
Chair of Computer Networks
Faculty of Computer Science
Technical University of Dresden
Dresden, Germany

Professor Peeter Ellervee
Department of Computer Engineering
Tallinn University of Technology
Tallinn, Estonia

Opponent

Professor Jan Madsen
DTU Informatics
Technical University of Denmark
Kongens Lyngby, Denmark

Abstract

This thesis proposes Hierarchical Agent-based Adaptation (H2A), a scalable platform-based design paradigm for the rapidly expanding parallel embedded systems. With the constant progress of semiconductor industry, the available hardware resources are steadily increasing, while the constraints in power/energy consumption and dependability issues pose tougher challenges than ever before. This phenomenon calls for a system-level paradigm shift towards self-aware and adaptive (SAA) systems, which are able to provide dynamic performance optimization based on self-monitored status and incidents.

The thesis motivates the paradigm shift towards SAA systems by examining the technology trends in parallel computing and embedded systems. Then it explores the existing techniques for run-time monitoring and reconfiguration, either with centralized, clustered or purely distributed architectures. The design and integration of these existing techniques and potential mechanisms require a scalable management backbone to provide global and local services, in particular for energy management and dependable computing. Answering this need, the thesis introduces the H2A system architecture, which integrates self-adaptation controllers, *Agents*, for monitoring and reconfiguring systems on hierarchical levels. The thesis overviews the functional partition of agents for coarse- and fine-grained services, and formulates the software/hardware (SW/HW) co-synthesis guidelines to implement the agents in a scalable manner.

To demonstrate the design of agents for run-time adaptation, the thesis first presents hierarchical energy management service. Energy-aware mapping is performed on the top-level platform agent. Cluster-level energy management is designed as clustered decision-making and distributed reconfiguration (CDDR), which is jointly contributed by both the cluster and cell agents. To support CDDR, hybrid monitoring of local traffic loads and application mile-

stones is performed by each cell agent. In addition to the energy management services, the thesis presents the design of H2A architecture for dependable computing. In particular, an approach for dynamic clusterization is proposed, where processing elements can be flexibly organized into different clusters lest run-time failures or performance degradation. Three-level supporting structures and a full-mesh-based physically separate monitoring network are designed to enable the dynamic clusterization. The supporting structures can be extended for other run-time adaptation services, such as energy management, thus achieving a portable and compatible system architecture.

As a proof-of-concept for the H2A design paradigm, a RTL(register transfer level) implementation of Self-Aware and Adaptive Network-on-Chip (SAA-NoC) is presented. The platform agent is implemented as software running on a dedicated Leon processor. Each cluster agent integrates a software thread running on a processor and a hardware-based CLT (cluster look-up table). Each cell agent is implemented as hardware circuits wrapped within one NoC node. Both hierarchical energy management and dynamic clusterization services are implemented. Experiment results validate that the H2A system architecture can fully exploit coarse- and fine-grained adaptation services to reduce the energy consumption under the influence of unpredictable errors, while meeting both system and local performance requirements. The overhead analysis shows that both software and hardware overheads are minimal and scalable for the agent subsystem.

Acknowledgements

Writing this doctoral thesis, ever since the beginning of my PhD program, has not been an easy process. I would like to firstly express my gratitude to my supervisors, Prof. Hannu Tenhunen, Prof. Jouni Isoaho, Prof. Juha Plosila and Prof. Pasi Liljeberg. Prof. Tenhunen has inspired and encouraged me to explore ambitious research topics, and provided me with high-level guidance towards my academic dream. Prof. Isoaho constantly encourages me to tackle challenging issues, by experimenting unconventional approaches. The H2A design paradigm originates from his vision, which I feel a true privilege to work on. Prof. Plosila has always been a dedicated and supportive supervisor in my research. We have together worked on many research tasks, and I am grateful for his support and encouragement of my sometimes impulsive ideas and plans. Prof. Liljeberg has given me valuable help through commenting and advising on topics and publication contents. We have also cooperated on several research activities, which are appreciated learning experiences in my PhD program. In addition, I wish to specially thank Prof Axel Jantsch from Royal Institute of Technology, Sweden. I have been under his guidance ever since my master thesis. Prof. Jantsch is a researcher and scientist I have been constantly learning from. I am grateful for all the discussions and cooperation we have had, which inspire my passion to research and science.

In addition to my supervisors, I would like to sincerely thank my fellow colleagues. Without their generous help, it is impossible to make progress on the challenging research topic. Foremost, Dr. Ethiopia Nigussie has been a inspiring guide, cooperative partner and loyal friend. My research progress has always been facilitated by her insights, comments and encouragement. It has been a true pleasure to have her as a friend and colleague. Besides, Khalid Latif, Kameswar Rao Vaddina, Syed Asad Jafri and Bo Yang have all given me highly-appreciated help and cooperation in my research path.

Our discussion and joint works inspire each other's study and research. In particular, I would like to sincerely thank Syed Asad Jafri for his cooperation in the implementation of the H2A design paradigm. His expertise and dedication have enabled the concrete demonstration of the novel paradigm.

I would like to thank three external experts for reviewing my thesis and being the opponent for my thesis defence: Prof. Jan Madsen from Technical University of Denmark, Prof. Waltenegus Dargie from TU Dresden, and Prof. Peeter Ellervee from Tallinn University of Technology. Thanks to their insightful comments, the thesis has been modified and improved.

I wish to thank all the colleagues and coworkers in Embedded Computer and Electronic Systems (ECES) lab and Communication Systems (ComSys) lab of Turku Center for Computer Science (TUCS). My research during these four years has been helped by many researchers and fellow students. Working in this environment has been an interesting and fruitful experience.

I would like to acknowledge the financial support from GETA (the Graduate School in Electronics, Telecommunications and Automation), which has been the main funding source of my PhD program. In addition, I would like to thank the support from Nokia Foundation, Wäinö Edward Miettinen and Ulla Tuominen Foundation.

Last but definitely not least, my most gratitude goes to my family. My mother XiaNan and father XiangChong have been supportive and understanding of my career and life choices. Any humble achievement I have made is owing to your nurturing and encouragement. In addition, I would like to thank my best friend Janne Alanen for all his caring and support of my life in Finland.

Contents

Abstract **iii**

Acknowledgments **iii**

List of Publications **ix**

List of Figures **xii**

List of Tables **xvi**

1 Introduction **1**
1.1 Era of Massively Parallel Computing 1
1.2 Design Challenges of Parallel Embedded Systems 3
1.2.1 Energy Efficiency 4
1.2.2 Dependable Computing 5
1.2.3 Scalability 6
1.3 Thesis Objectives 8
1.4 Thesis Contributions 9
1.5 Organization of Thesis 10

2 Paradigm Shift towards Self-Aware and Adaptive Systems **12**
2.1 SAA Systems 12
2.2 Monitoring and Reconfiguration Operations 15
2.3 Paradigm Evolution towards 3-Dimensional Design Space . . . 18
2.4 Chapter Summary 21

3 Design Exploration of Self-Adaptation Architectures **22**
3.1 Monitoring, Decision-Making and Reconfiguration Processes . . 23
3.1.1 Monitoring 24

3.1.2 Decision-Making 24
3.1.3 Reconfiguration 26
3.1.4 Monitoring Communication 27
3.2 Centralized, Clustered and Distributed Architectures 28
3.2.1 Physical Support for Voltage and Frequency Reconfiguration 28
3.2.2 Centralized Architecture 30
3.2.3 Clustered Architecture 31
3.2.4 Distributed Architecture 32
3.3 Quantitative Comparison 33
3.3.1 Platform Setting 33
3.3.2 Traffic Generation 34
3.3.3 Simulation Results 36
3.3.4 Performance 37
3.3.5 Energy Efficiency 38
3.4 Summarizing Comparison 41
3.5 Chapter Summary 41

4 Hierarchical Agent-based Design Platform 44
4.1 Agent as a Design Abstraction 44
4.2 H2A Architecture 46
4.3 Agent Implementation Guidelines 49
4.3.1 Alternatives 51
4.3.2 SW/HW Co-Design for Agents 52
4.4 Agents and Operating System 53
4.5 Related System Architectures 55
4.6 Chapter Summary 56

5 Coarse- and Fine-Grained Energy Management 58
5.1 System Architecture 58
5.2 Energy-Aware Application Mapping 62
5.3 Intra-Cluster Energy Management 64
5.3.1 Cluster Architecture Overview 65
5.3.2 Load and Latency Monitoring 66
5.3.3 Clustered Decision-Making, Distributed Reconfiguration 68
5.4 System Integration and Quantitative Evaluation 69
5.4.1 Experimental Setting 69

5.4.2 Results 71
5.4.3 Implementation Discussion 76
5.5 Chapter Summary 78

6 Dynamic Clusterization for Dependable Computing 80
6.1 System Architecture Overview 80
6.2 Three-Level Supporting Structures 83
6.3 Inter-Agent Monitoring Network 85
6.4 System Integration 92
6.5 Further Discussion 94
6.6 Chapter Summary 94

7 Implementation of Self-Aware and Adaptive NoCs 96
7.1 NoC Platform 97
7.2 Three-Level Agents and Supporting Structures 99
7.2.1 Platform Agents 100
7.2.2 Cluster Agents and CLT 102
7.2.3 Cell Agents and CIR 105
7.3 System Integration 105
7.4 Experiments 108
7.4.1 Experimental Setting 109
7.4.2 Energy Management Results 111
7.4.3 Fault-Tolerance Result 114
7.5 Overheads 117
7.6 Chapter Summary 118

8 Conclusion 120
8.1 Design Era of SAA systems 120
8.2 Hierarchical Agent Architecture 121
8.3 Self-Aware and Adaptive NoC Implementation 122
8.4 From Parallel to Distributed Computing: Future Work 123

References 125

A HLS-DoNoC: High-Level Simulator for Dynamically Organizational NoCs 140
A.1 Modular Overview 141

A.2 Network Kernel 142
A.3 Simulating Dynamic Clusterization 142
A.4 Simulating Cluster-based Reconfiguration 143
A.5 Integrating Power Models for DoNoCs 145
A.6 Traffic Generation 145
A.7 Simulator Feature Summary 146

List of Major Publications

The thesis contains the research in the following publications with new unpublished material (in Chapter 5, 6 and 7).

1. Liang Guang, Ethiopia Nigussie, Juha Plosila, Jouni Isoaho, Hannu Tenhunen, Survey of Self-Adaptive NoCs with Energy-Efficiency and Dependability, International Journal of Embedded and Real-Time communication Systems (IJERTCS), 3(2), pp.1-22, 2012.

2. Liang Guang, Ethiopia Nigussie, Juha Plosila, Hannu Tenhunen, Dual Monitoring Communication for Self-Aware Network-on-Chip: Architecture and Case Study, International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS), 3(3), pp.73–92, 2012.

3. Liang Guang, Juha Plosila, Jouni Isoaho, Hannu Tenhunen , HAMSoC: A Monitoring-Centric Design Approach for Adaptive Parallel Computing , Chapter in book "Autonomic Networking-on-Chip: Bio-inspired Specification, Development and Verification", editor Phan Cong-Vinh, Taylor & Francis/CRC Press, Florida, USA, pp.135-164, 2012.

4. Liang Guang, Ethiopia Nigussie, Pekka Rantala, Jouni Isoaho, Hannu Tenhunen, Hierarchical Agent Monitoring Design Approach Towards Self-Aware Parallel Systems-on-Chip. ACM Transactions on Embedded Computing Systems (TECS) 9(3), pp.25(1-24), 2010.

5. Liang Guang, Ethiopia Nigussie, Jouni Isoaho, Pekka Rantala, Hannu Tenhunen, Interconnection Alternatives for Hierarchical Monitoring Communication in Parallel System-on-Chip. Microprocessors and Microsystems (Elsevier) 34(5), pp.118-128, 2010.

6. Liang Guang, Ethiopia Nigussie, Juha Plosila, Jouni Isoaho and Hannu Tenhunen. Coarse and Fine-Grained Monitoring and Reconfiguration

for Energy-Efficient NoCs. Proceedings of International Symposium on System-on-Chip, October, 2012, to appear.

7. Liang Guang, Ethiopia Nigussie, Juha Plosila, Jouni Isoaho, Hannu Tenhunen. HLS-DoNoC: High-Level Simulator for Dynamically Organizational NoCs, In Proceeding of IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, pp.89-94, 2012.

8. Liang Guang, Syed Asad, Tony Yang, Juha Plosila and Hannu Tenhunen, Embedding Fault-Tolerance with Dual-Level Agents in Many-Core Systems, 1st Median Workshop, pp.41-44, 2012

9. Liang Guang, Bo Yang, Juha Plosila, Jouni Isoaho, Hannu Tenhunen, Hierarchical Agent Monitoring Design Platform towards Self-Aware and Adaptive Embedded Systems. In: César Benavente-Peces, Joaquim Filipe (Eds.), Proceedings of International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2011), pp.573 - 581, SciTePress, 2011.

10. Liang Guang, Bo Yang, Juha Plosila, Khalid Latif, Hannu Tenhunen, Hierarchical Power Monitoring on NoC-A Case Study for Hierarchical Agent Monitoring Design Approach. In proceedings of IEEE Norchip 2010, 1-6, IEEE, 2010.

11. Liang Guang, Pasi Liljeberg, Ethiopia Nigussie, Hannu Tenhunen, A Review of Dynamic Power Management Methods in NoC Under Emerging Design Considerations. In Proc. of IEEE Norchip 2009, 1-6, IEEE, 2009.

12. Liang Guang, Ethiopia Nigussie, Lauri Koskinen, Hannu Tenhunen, Autonomous DVFS on Supply Islands for Energy-Constrained NoC Communication. In: M. Berekovic et al. (Ed.), Proceedings of ARCS (International Conference on Architecture of Computing Systems)09 Conference, LNCS 5545, 183-194, Springer-Verlag Berlin Heidelberg, 2009.

13. Liang Guang, Ethiopia Nigussie, Hannu Tenhunen, System-Level Exploration of Run-Time Clusterization for Energy-Efficient On-Chip Communication. In Proc. of 2nd international workshop on Network-on-chip architectures, 63-68, ACM, 2009.

14. Liang Guang, Pekka Rantala, Ethiopia Nigussie, Jouni Isoaho, Hannu Tenhunen, Low-Latency and Energy-Efficient Monitoring Interconnect for Hierarchical-Agent-Monitored NoCs. In Proceedings of IEEE Norchip 2008, 227 - 232, IEEE, 2008.

List of Figures

1.1 An 8*8 Network-on-Chip in TILE64 processor [8] 3

2.1 Functional Overview of Self-Aware and Adaptive Systems . . . 14
2.2 Self-Aware and Adaptive Systems: A Particular Perspective on Autonomic Computing. 15
2.3 Conventional Model-based Design vs. Platform-based Design . 19
2.4 SAA as a Separate Design Dimension 20

3.1 Monitoring, Decision-Making and Reconfiguration Processes in Self-Adaptation Architecture 23
3.2 Illustrating Centralized and Distributed Decision-Making on NoCs 25
3.3 Overheads of Fine-grained Reconfiguration for Individual Voltage and Frequency Domains 27
3.4 Voltage Regulator based Energy Management 29
3.5 Multiple On-Chip Power Delivery Network based Per-Core Voltage Switching 30
3.6 Centralized Energy-Management Architecture 30
3.7 Cluster-based Decision-Making and Reconfiguration 31
3.8 Distributed Energy Management with MPN-based Physical Support 32
3.9 An Example of B-model Traffic (b=0.6) 35
3.10 Experimental Platform Running Four Traffic Traces (each trace runs in a 4*4 cluster) 36
3.11 Temporal Latency of Four Traffic Traces in Different Energy Management Architectures 37
3.12 Number of Switchings in Different Energy Management Architectures 39

3.13 Average Communication Energy of Four Traces in Three Energy Management Architectures 40
3.14 Energy-Delay Product of Four Traces with Different Energy Management Architectures (*CE*, *CL* and *DI*) 40

4.1 The Integration of Agents for Self-Adaptation on Processing and Communication Platforms 45
4.2 Multi-layered Agents for Hierarchical Monitoring, Decision-Making and Reconfiguration 47
4.3 Application Awareness with Program Heartbeats 48
4.4 Agents and Operating System 54

5.1 Hierarchical Agents for Energy Management upon Distributed Reconfigurable Platform 60
5.2 Exemplified Two-Step Energy-Aware Mapping Algorithm on Platform Agent 63
5.3 Overview of CDDR Technique for Intra-Cluster Energy Management 65
5.4 Relation between Buffer Load and Latency (in Cycles) of Three Traffic Traces in Different Time Windows 67
5.5 Monitoring Load in Each Router 67
5.6 Latency Monitoring with Cluster and Cell Agents (exemplifying one MIR for brevity 68
5.7 Flow Diagram of Clustered Decision-Making with Hybrid Monitoring 70
5.8 Mapping Two Applications on Four Clusters in a 6×6 NoC . . 72
5.9 Energy Consumption and Stream Latency of AUTO Application 74
5.10 Energy Consumption and Stream Latency of CON Application 77

6.1 Dynamic Clusterization for Fault-Tolerance and Performance Degradation 82
6.2 Monitoring, Decision-Making and Reconfiguration Stages in Dynamic Clusterization 84
6.3 RLT for Resource Tracing on Platform Agent 85
6.4 CLT for Currently Utilized Cells in Each Cluster 86
6.5 Mesh Network for Minimum-Distance Dimension-Order Routing 87

6.6 Interconnection Architectures for Inter-Agent Monitoring Communication . . . 90
6.7 Energy Consumption and Latency of Three Monitoring Communication Architectures (B: baseline, P: physically separate monitoring network, T: TDMA-based router architecture; (B,H), (B,U), (U,U), (U,H)): the four data traffic patterns. . . 91
6.8 Inter-Agent Communication Process for Dynamic Clusterization 93

7.1 Distributed Shared Memory NoC Platform with Fine-Grained Voltage and Clock Generation . . . 98
7.2 Software Implementation of Platform Agent and RLT . . . 101
7.3 SW/HW Co-synthesis of Cluster Agent . . . 102
7.4 CLT: A Reconfigurable Pre-Processing Hardware Unit for Cluster Agent . . . 104
7.5 Cell Agent and the Supporting Structures wrapped in a NoC Node . . . 106
7.6 Implemented Hierarchical Agent Services . . . 107
7.7 Interaction between Three-Level Agents on SAA-NoC (PLA: platform agent, CLA: cluster agent, CEA: cell agent, S: spare, DM: distributed memory, MIR is omitted for brevity) . . . 108
7.8 Format of Inter-Agent Monitoring Communication . . . 109
7.9 Mapping Two Applications with Six Scenarios on SAA-NoC (the number of processors is only for illustration purpose) . . . 110
7.10 Minimizing Individual Router Energy with Latency-Oblivious and Latency-Bounded Cluster-Level Energy Management . . . 112
7.11 Per-Stream Energy Consumption in Latency-Oblivious Cluster-Level Energy Management (MPEG) . . . 112
7.12 Per-Stream Energy Consumption in Latency-Oblivious Cluster-Level Energy Management (BASIZ) . . . 113
7.13 Energy-Aware Remapping for Dynamic Clusterization . . . 115
7.14 Per-stream Energy Consumption in Dynamically-Reconfigured Clusters (Latency-Oblivious Management) . . . 116

A.1 Module Overview of Simulation Framework . . . 141
A.2 Cluster Look-up Table and Run-Time Monitoring Communication for Dynamic Clusterization . . . 143
A.3 Simulating Ratiochronous Timing in DoNoC . . . 144

A.4 Integrating Power Models for Run-Time Voltage and Frequency Switching 146

List of Tables

1.1 Sources of Variations and Unpredictability in On-Chip Interconnects . . . 7

2.1 The Diversity of Monitoring and Reconfiguration Operations in Parallel Embedded Systems . . . 16

3.1 Average Flit Latency (Cycles) in Monitoring Network . . . 38
3.2 Summarizing Comparison of Centralized, Clustered and Per-Router DVFS in Energy-Performance Tradeoff . . . 42

4.1 Generic Functional Partition of Agents on the Platform . . . 50
4.2 Agent Implementation Guidelines . . . 52

5.1 Monitoring and Reconfiguration Techniques for Energy-Efficient NoCs . . . 59
5.2 Agents' Operations for Coarse- and Fine-Grained Energy Management . . . 71

6.1 Area Estimation (um^2) for Each Interconnection Architecture . . . 92

7.1 Voltage frequency pairs (the bold font labels the allowed frequencies based on GRLS clocking and the corresponding minimal voltages meeting the timing constraints) . . . 110
7.2 Per-Stream Energy and Latency in Latency-Bounded Cluster-Level Energy Management . . . 114
7.3 Injected Processor Failures and Utilized Spares (as intra-cluster locations) . . . 115
7.4 Comparison of Stabilized Energy and Latency in Fault-Injected and Fault-Free Clusters (Latency-Bounded Management) . . . 116
7.5 Overheads of Agents and Supporting Structures . . . 119

A.1 Enhanced Features for Simulating DoNoCs 147

List of Abbreviations

CDDR	Clustered Decision-Making, Distributed Reconfiguration
CIR	Cluster Identifier Register
CLT	Cluster Look-up Table
DVFS	Dynamic Voltage and Frequency Scaling
GALS	Globally Asynchronous Locally Synchronous
GRLS	Globally Ratiochronous Locally Synchronous
H2A	Hierarchical Agent-based Adaptation
MBD	Model-Based Design
MDR	Monitoring, Decision-Making, Reconfiguration
MIR	Milestone Instruction Register
NoC	Network-on-Chip
PBD	Platform-Based Design
PVT	Process, Voltage and Temperature (variations)
RLT	Resource Look-up Table
RT	Re-routing Table
RTL	Register Transfer Level
SAA	Self-Aware and Adaptive
SW/HW	Software/Hardware
VLSI	Very Large Scale Integration

Chapter 1

Introduction

To achieve high computing performance with reduced power consumption, massively parallel processing has become a mainstream computing platform [4]. The constant scaling of CMOS technology enables the integration of an increasing number of components onto a single die, leading to multi-core and many-core on-chip computing [11]. Meanwhile, considerable deep submicron effects, such as variations, leakage currents and noises, strongly challenge the embedded system design. Self-adaptive computing [99] is a novel paradigm to provide the system with superior performance, efficiency and variability compared to conventional static worst-case-based design. This thesis presents a design platform, Hierarchical Agent-based Adaptation (H2A), to realize Self-aware and Adaptive (SAA) parallel embedded systems.

1.1 Era of Massively Parallel Computing

Parallel and distributed computing has become one major focus of research community in computer science and engineering [4]. Parallel computing provides much higher performance than sequential processors. For instance, TeraFLOPS [119], an 80-core processor, achieves over 1.0 TFLOPS (10^{12} floating-point operations per second). The Tile64 [8] processor is able to execute 192 billion 32-bit operations per second at $1GHz$. As the processing engines for a large amount of data, massively parallel processors are expected to serve in a wide variety of civil and military applications, including (but surely not limited to) multi-media processing, scientific applications, large data centers and personal computers or servers.

The promise of parallel computing is supported by the technology development, in particular in the semiconductor industry. The CMOS transistor size is constantly shrinking. In 2011, the DRAM pitch reaches $40nm$, and will reach $25nm$ in 2015 [30]. With the progress of semiconductor industry, the technological feasibility of massively parallel on-chip processing has been demonstrated by many industrial examples, including Intel 80-core TeraFLOPS processor [119], 48-core SCC (Single-chip Cloud Computer) [48] and Tilera 64-core TILE64 processor [8].

The many-core multi-processor is one primary form of parallel systems. By integrating a large number of simple processors, high performance can be achieved by parallelizing the applications while keeping the voltage lower to reduce the power consumption [4, 10]. The generic architecture does not exclude the utilization of dedicated accelerators, for instance, graphic engines, which are needed to perform specific processing with better efficiency than general-purpose processors [118].

One major challenge in many-core multi-processor is the communication network. For one thing, the large amount of parallel data processing may incur intensive communication volume [122]. The communication network is required to provide high bandwidth to avoid becoming the bottleneck of system performance [26]. In modern technology, the interconnect delay is much larger than the processing delay (a $9:1$ ratio in 2010 [Dally:2002a]). In addition, the power consumption of interconnect is significant compared to that of data processing. For instance, transferring 32-bit data across a $10mm$ chip consumes $17pJ$ in $50nm$ CMOS, while a 32-bit ALU operation only consumes $0.3pJ$ [25]. Thus, how to provide high-bandwidth, low-power communication is an essential design challenge, the so-called interconnect-centric design [94], in many-core parallel systems.

The Network-on-Chip (NoC) [54] has been proposed as a scalable architecture for high-performance many-core systems. It represents a new design methodology, which separates the design concerns of communication from computation [94]. In terms of the communication performance, NoCs achieve much higher bandwidth than conventional bus-based systems. Instead of reserving a whole channel for a single communication, NoC allows the simultaneous utilization of different network segments and/or timeslots for a large number of communication flows, thus providing much better throughput and scalability. In addition, most NoC architectures adopt modularized architec-

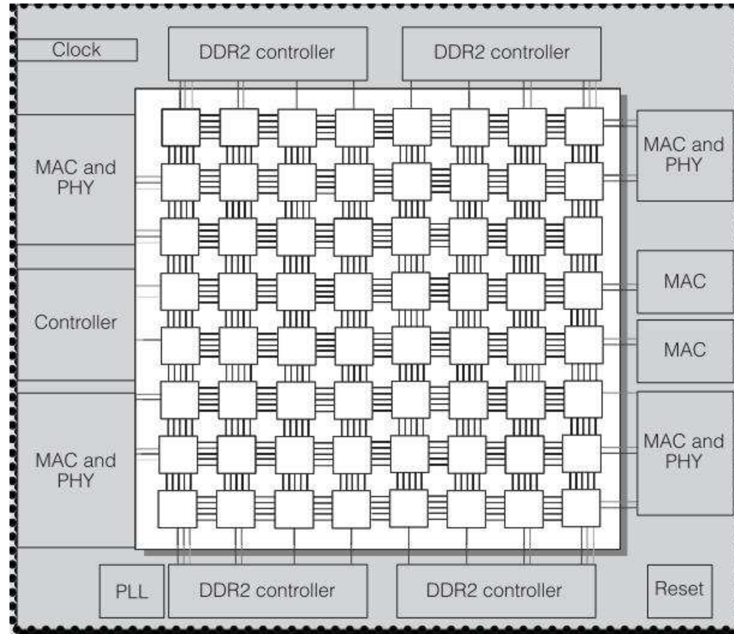


Figure 1.1: An 8*8 Network-on-Chip in TILE64 processor [8]

tures, in order to provide reusability and predictability [94]. Thus the NoC architecture can be applied to different parallel applications of various sizes. Due to these benefits, NoCs have been widely used for many-core VLSI systems, for instance, in TeraFLOPS processor [119] and TILE64 processor [8] (Fig. 1.1). In this thesis, the term *NoCs* is generally referred to NoC-based many-core systems, unless specifically stated otherwise.

1.2 Design Challenges of Parallel Embedded Systems

The parallel embedded system is an important and widely applied domain of parallel computing. Embedded components are extensively used for controlling, processing, and storage purposes in a large variety of applications, such as automotive, consumer electronics, spacecrafts and electronic medical devices. Compared to other parallel processing domains (e.g., supercomputers), embedded systems are more limited in hardware and software resources, and have specific requirements, e.g., real-time constraints or high reliability [88]. In addition, commercial embedded systems and products have strict time-to-

market constraints. This thesis is mostly concerned with the following three aspects of design challenges:

1.2.1 Energy Efficiency

Power and energy consumption is the primary design concern for resource-limited embedded systems. On one hand, many embedded systems have limited power provision, e.g., portable battery-charged devices. On the other, the cooling capacity of embedded systems is lower than, e.g., desktop applications (with a heat sink on package [31]). While the transistor density is consistently increasing, the power budget remains flat. As analyzed by [118], future multi-core smartphones need to support $100GOPS$(i.e., 10^{11} operations per second) with $1W$ power budget. Clearly how to efficiently utilize the limited power and energy resources for an ever increasing computation capacity will pose a major challenge to parallel embedded systems.

Instead of being a unified goal, power and energy consumption actually concerns several related but different metrics. Power consumption can be measured as the temporary peak power or the average power during a period. Energy is the integration of power over a period of time ($Energy = \int_{t=t0}^{t=t1} Power(t) \times dt$). Energy can be measured as energy per operation or the total energy consumption of the system in a period of time. Both power and energy can be divided into the dynamic part and the static part. Dynamic power is consumed by the switching of capacitances, and only exists when there are activities on the datapath, control path or the clock tree. Static power is consumed by leakage in the semiconductor materials, in particular subthreshold and gate leakage [61]. Static power consumption exists as long as the circuits are powered on. Power and energy in a many-core system are contributed by all components, including processors, networks, memories and I/O interfaces. With an increasing number of high-speed on-chip interconnects, the network contributes a significant part of power and energy consumption. For instance, TeraFLOPS reports 28% power consumption attributed to the on-chip communication [119]. Thus minimizing the power and energy consumption of the on-chip network is one of the major low-power methods in parallel embedded systems [107].

Different power and energy metrics are considered in various design scenarios. The energy consumption is usually considered for battery-charged portable devices, for instance, mobile phones or digital players. The average

power consumption is a general metric for comparison of different systems, and is also used to determine the required cooling capacity. The peak power gives the requirement on the maximal driving capacity of the power supply. The power-aware and energy-aware design may have a strict power budget, e.g., for a system with stringent cooling capacity. In other cases, a best-effort design is followed to reach as low power and energy consumption as possible.

Among different metrics, energy efficiency is of primary interest to the embedded systems. It indicates how much energy is consumed to achieve the expected performance with either hard or soft constraints. For a real-time system with hard deadlines, the energy should be minimized while the performance deadline is met. On the other hand, for scenarios without hard constraints, the design is usually made towards as low energy consumption as possible. For instance, [77] presents algorithms to reduce the energy consumption and the execution time with a best-effort approach. This thesis addresses maximizing energy efficiency as one of the primary design targets.

1.2.2 Dependable Computing

Dependability is an integral concept that encompasses the attributes of availability, reliability, safety, integrity and maintainability [6]. A dependable system should provide the expected performance under different situations, so that the system can be trusted [6]. In general-purpose many-core systems that may potentially host diverse applications, the dependability issues become more challenging due to the different requirements of applications and heterogeneity of components.

Hardware variations pose one major threat to dependable computing in deep-submicron era. Different sources of variations are generalized as PVT (Process, Voltage and Thermal) variations [9], which significantly affect the system performance and dependability.

Process variations refer to the deviation of actual physical parameters from nominal values due to the manufacturing process or post-manufacture aging. Such parameters include geometry sizes (e.g., effective transistor size L_{eff}, metal line width and thickness, etc.) and material parameters (e.g., the level of doping in the semiconductor) [121]. With technology scaling, the process variation is increasing. For instance, the deviation of L_{eff} increases from 32% in 1997 to 47% in 2006 [5]. Voltage variation refers to the difference of voltage values, in particular the supply voltage, on different places in the power deliv-

ery network [44]. The aggressive voltage scaling to reduce power consumption makes the nominal voltage very low, thus the effect of voltage drop is more severe than before [44]. As the supply voltage determines the propagation delay of interconnects, its variation will affect the communication speed of on-chip networking. Thermal variations refer to the different temperatures at various places in a VLSI chip [106], which are also changing with time. The temperature not only affects the stability of circuits, but also influences the leakage power [69].

PVT variations lead to unpredictable errors at the run-time. For one thing, the variations lead to more hardware defects [60], if the deviation is so large as to cause short or open circuit. In addition, the variations may lead to timing errors [103]. For instance, large voltage drop causes the supply voltage to be lower than the nominal value, thus the circuits run slower than expected. Moreover, temperature variations may lead to inaccurate estimation of power consumption due to the leakage power [69].

In addition to PVT variations, other deep-submicron effects also influence the dependability of the circuits and systems, such as crosstalk, radiation-induced errors and NBTI (Negative bias temperature instability). Crosstalk refers to the noise between adjacent wires. Radiation-induced faults [16] are caused by particle hits in the silicon substrate, which are strong enough to flip the values stored in the latches. NBTI causes the shift of the threshold voltage. With technology scaling, the distance between wires becomes smaller, and the node capacitance decreases, thus the circuit is more susceptible to crosstalk and radiation-induced errors [16].

On-chip interconnects, among other components in parallel embedded systems, suffer from the PVT variations and other deep-submicron effects. As summarized by Table 1.1, the propagation delay of on-chip communication is affected by the process and voltage variations. Its reliability is influenced by the permanent or transient errors of various causes.

1.2.3 Scalability

With the size and complexity of parallel systems dramatically increasing, the scalability of design methods and system architectures becomes essential. This is in particular important for embedded systems, due to the time-to-market requirements. Scalability is also a broad topic, whose specific criteria are dependent on the design tasks. Generic criteria for scalability can be found

Table 1.1: Sources of Variations and Unpredictability in On-Chip Interconnects

Source	Influence on On-chip Interconnects
Physical geometry variations	physical defects (permanent failure), threshold voltage deviation (causing change of delay and leakage power), clock skew mismatch
radiation/ soft-error	transient data error
crosstalk	transient data error, delay variation [85]
voltage variation	propagation delay variation, timing error
thermal variation	leakage power variation [69]

in [114], which include: 1) the system can integrate an increasing number of components; 2) the components can be located in geographically distant locations; 3) the administration of the system can still be convenient with more components and larger system size. The second requirement is related to physically distributed computing, which is beyond the scope of this thesis. In the context of parallel systems, the following three aspects of scalability are of particular interests:

- Performance scalability. As the classic requirement of parallel systems, the performance scalability is often quantified by the speedup of the same problem on a parallel computing compared to a single processor [66]. For applications that require an extensive amount of data processing, for instance, scientific computing, the speedup as a performance scalability metric is highly important. However, in the embedded computing domain, we concern much less for the speedup. For one thing, many applications, for instance, multimedia processing, have real-time requirements, thus increasing the processing speed beyond the requirement is meaningless. For another, embedded systems are resource limited, thus achieving higher speedup without considering the incurred overhead is often unfeasible in the practical implementation.

- Overhead scalability. As the intrinsic constraints of embedded computing system, area and power/energy overheads are major limiting factors

to the scalability. The area overhead is a conventional concern especially for portable embedded systems. With the constant increasing of transistor densities and multi-metal-layer fabrication technology, the power or energy consumption has replaced area as the primary constraint [95]. Generally speaking, the software and hardware resources should incur linear or sub-linear power and area overheads when more functions and components are integrated into the system.

- Design effort scalability. Embedded systems have stringent time-to-market requirements, thus the scalability in terms of the design effort is essential to the system development. With more components integrated into a complex parallel embedded system, the involved design effort should be minimized to keep a reasonable development cycle. Design orthogonalization and design reuse are system-level principles to reduce the design effort, as embodied in the state-of-the-art Platform-based Design (PBD) paradigm [58, 100]. In particular, the design process should be performed on a high-level abstraction that hides the unnecessary low-level details. The components should be modularizable thus easily integrated. The system architecture itself should be generically applicable to different applications and support a wide diversity of functions.

1.3 Thesis Objectives

As embedded systems become highly parallel and distributed, the design of such systems incurs overwhelming complexity to deal with the variations, power budget and dependability issues. Static design and external reconfiguration become either infeasible or inefficient, since the run-time system status is often unpredictable and fast changing. The thesis aims to develop a scalable approach for the design of parallel embedded systems, which can autonomously achieve the optimization of performance, power consumption and dependability under the variations of workloads and system status.

To achieve the goal of this thesis, firstly, the paradigm shift towards self-aware and adaptive (SAA) computing in embedded system domain will be motivated. Self-aware and adaptive properties enable the system to optimize itself by observing the internal and external status. Even though the topics of self-adaptive computing has been discussed in software engineering domain

[99], its application in embedded computing systems is still in the emergent stage. The justification of such paradigm shift is required from the perspectives of current technology challenges and potential design benefits.

To enable the aforementioned paradigm shift, a generic and scalable architecture for SAA computing in parallel embedded systems needs to be proposed. Most existing works utilize run-time management techniques to optimize specific performance metrics. However, ad-hoc approaches incur design complexities when the system expands or the techniques are ported to a different platform. Instead, a generic system architecture, where a wide diversity of self-aware and adaptive services can be provided, is able to exploit design reuse and offer scalability in terms of the design effort (Section 1.2.3).

To demonstrate the proposed architecture, a proof-of-concept prototype is needed. The implementation should prove the feasibility of the architecture in the current or emerging technology. SAA features should be experimented on the implementation, with practical applications. Measurement and analysis of concrete performance results, for instance, power, energy and area, are needed to show the power efficiency, dependability and scalability of the system.

1.4 Thesis Contributions

To reach the objectives, the thesis develops H2A (Hierarchical Agent-based Adaptation) design paradigm. The work contributes to the state-of-the-art embedded system design with the following achievements:

1. SAA computing is introduced and motivated in the embedded system domain. The thesis identifies monitoring and reconfiguration services as the main enabler of SAA computing. It presents an abstract model of the SAA system. To realize such systems with design efficiency and scalability, the thesis proposes the orthogonalization of monitoring and reconfiguration services as a dedicated design dimension. Such orthogonalization is reasoned from the perspective of design reuse of existing computation and communication platforms.

2. The thesis develops H2A architecture for SAA systems, where agents on different organizational levels make a joint effort to monitor and reconfigure the system. The functional partition among agents is studied. The SW/HW(software/hardware) co-design guidelines of agents are formu-

lated to provide physical scalability. The design process from functional partition, architectural design to microarchitecural implementation is presented. On an architectural-level network simulator, extensive simulations are performed to demonstrate the benefits and efficiency of agent operations. Examples are given for hierarchical energy management and run-time dynamic clusterization for dependable computing in NoCs.

3. Self-adaptive NoC is implemented as a proof-of-concept for H2A architecture. Functions of different levels of agents and the inter-agent communication are implemented on Nostrum [65] NoC with Leon3 processors and distributed shared memory. Various monitoring services, including hierarchical energy management and dependability-driven clusterization, are realized with SW/HW co-design and synthesis. A cycle-accurate RTL (register transfer level) simulator is utilized to experiment on the self-adaptive NoC with representative applications. Performance, energy consumption and area overhead are analyzed.

1.5 Organization of Thesis

The thesis is composed of eight chapters. Following introduction, Chapter 2 motivates the paradigm shift towards SAA systems, in the domain of parallel embedded computing. It achieves the first objective of the thesis. The thesis will discuss the definition SAA systems, and then overview the current monitoring and reconfiguration techniques in parallel embedded computing. In order to reduce the design complexity for scalable system development, it proposes a dedicated design layer for adaptive services.

Chapter 3 to 6 address the second objective of the thesis, i.e., a generic and scalable architecture for SAA computing. Firstly Chapter 3 examines the current monitoring and reconfiguration architectures in parallel embedded systems, including centralized, clustered and distributed architectures. Simulations are performed on energy management in NoCs. Based on this study, the pros and cons of different architectures can be identified. And the need of a systematic approach for various levels of services is reasoned. Chapter 4 introduces the H2A design paradigm, starting with an overview of the agent concept and the hierarchical organization of agents. To provide the scalability in terms of agent overheads, it proposes implementation guidelines for each level of agents. Chapter 5 presents a concrete service, energy management,

with H2A in NoCs. Energy-aware mapping and clustered decision-making with distributed reconfiguration are designed as coarse- and fine-grained services on three levels of agents. Simulations are performed to demonstrate the efficiency and effectiveness of the energy management architecture. Chapter 6 presents the enhancement of agent architecture to provide dependable computing with dynamic clusterization.

Chapter 7 achieves the third objective of the thesis by presenting a proof-of-concept implementation of H2A design on NoCs. Based on the Nostrum NoC architecture, with Leon 3 processors and distributed shared memory, software and hardware agents are implemented. The software agent (platform agent) is realized on a dedicated Leon processor. The cluster agent is implemented as a software thread running on a processor and a hardware-based Cluster Look-up Table (CLT). The cell agent is implemented as hardware circuits attached to each NoC node. The implementation is done at register transfer level, and the results in terms of performance, energy consumption, area and timing overhead are analyzed.

The thesis is concluded in Chapter 8 with a summarizing presentation of the status quo of H2A design paradigm and a visionary discussion of its future application in a wider scope of parallel systems.

Chapter 2

Paradigm Shift towards Self-Aware and Adaptive Systems

This chapter motivates the paradigm shift towards Self-Aware and Adaptive (SAA) systems in the embedded computing domain. The continuous development of technology calls for a more efficient manner of dealing with complexity, which necessitates autonomous management of computing systems. For embedded systems, the needs for better performance, dependability and power consumption are addressed by diverse monitoring services and the corresponding reconfiguration operations. To design monitoring services in a scalable and portable manner, a dedicated design layer for SAA properties is proposed to separate the design concerns and reuse existing computation/communication architectures.

2.1 SAA Systems

The technology development of computing systems call for SAA properties. The concept originates from the classic Autonomic Computing [47]. The definitions of SAA have been discussed by some existing works in various contexts [99, 51] with similar explanation and wordings. This thesis defines a SAA embedded system as

> *One that is monitoring its own state and the environment in order to achieve the expected performance under potential envi-*

ronmental changes. The performance refers to both functional (e.g., execution time) and non-functional metrics (e.g., energy efficiency), either as hard constraints (e.g., power budget) or soft requirements (e.g., with as low power as possible).

The profound variations in system status and the increasing system complexity call for SAA properties. As discussed in Section 1.2.2, circuits and systems in modern technology suffer from various variations. Thus the system characteristics, for instance, path-delay/performance, follow statistical distribution [120]. The conventional worst-case design leaves a big design margin to accommodate the worst-case scenarios, which leads to extra power consumption or low performance. [95] proposes that the system should be adaptively adjusted based on internal and external conditions, the so-called *Always-Optimal* design. In addition to variations, the complexity of computing systems in general calls for autonomous management, as the proliferation of applications and increasing requirements for computing performance, efficiency and dependability pose overwhelming challenges to development [47].

The functional overview of a SAA system can be illustrated by Fig. 2.1. With the objectives determined by application scenarios, the system traces parameters from the application, the environment and its own resources. Given the observed information, it models the state of the application, the environment and its own resources using cost functions. Cost functions are preconfigured or dynamically reconfigurable models used for evaluating the states of the relevant entities. The results returned by the cost functions are used to determine if any actions are needed. Key objectives for a SAA system are dependability, scalability (so that the system can manage any number of resources) as well as power and energy efficiency (so the power consumption of the system can be optimized when other objectives are achieved). Major actions include logic reconfiguration (for instance, new network structure), physical reconfiguration (for example, new voltage supply), as well as outputs to external environment. In a parallel system, each component may be a SAA entity, and the cooperation of all entities should result in the general self-awareness and adaptation of the whole system.

Compared to the classic concept of autonomic computing with the four aspects of self-management (self-configuration, self-optimization, self-healing and self-protection [57], the notion of self-aware and adaptiveness focuses on the two distinct phases, awareness and adaptation, which are pillar processes

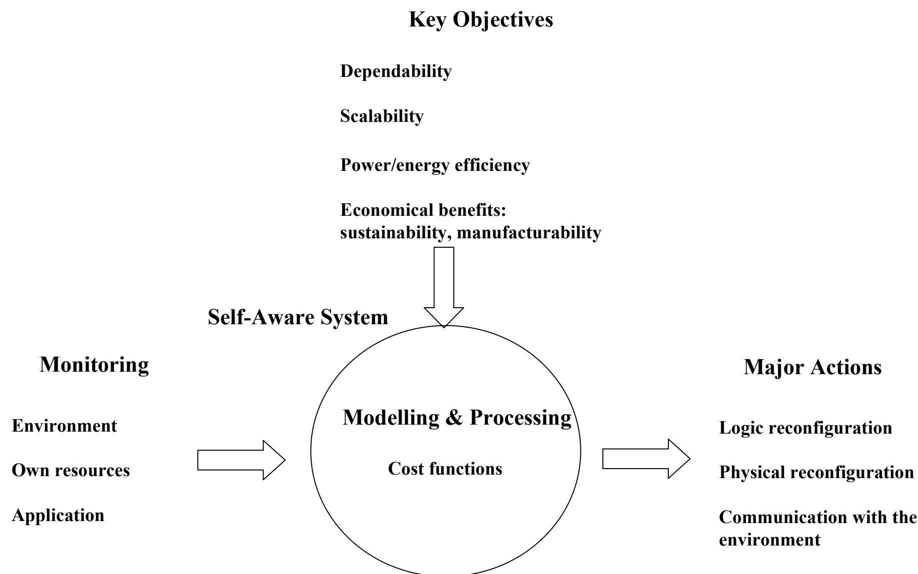


Figure 2.1: Functional Overview of Self-Aware and Adaptive Systems

in any type of closed-loop self-management. As illustrated in Fig. 2.2, awareness is the prerequisite for effective reconfiguration. A platform with a generic support for adaptation triggered by self-awareness is able to provide different types of autonomic operations, given proper configuration of the algorithms and microarchitectures. Specific to embedded computing, our research is focused on the system architecture to trace and reconfigure proper parameters with scalability in cost and design efficiency.

Compared to another closely-related concept, reconfigurable computing, SAA computing again emphasize on the architectural integration from observable parameters to corresponding reconfiguration. Reconfigurable computing, on the other hand, explores the vast design space of possible reconfiguration techniques on each implementation level. The SAA system utilizes the reconfigurable technologies as needed by the design task. For instance, when the system notices that its performance is worse than expected, it may seek adaptation to improve the performance by reconfiguring the FPGA-based processing fabric.

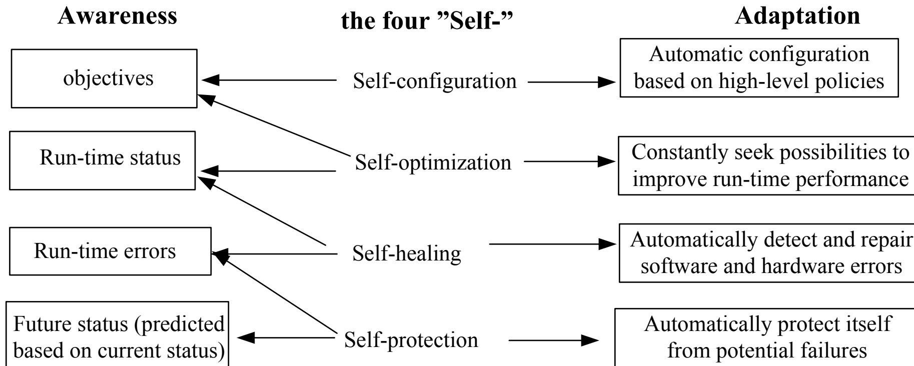


Figure 2.2: Self-Aware and Adaptive Systems: A Particular Perspective on Autonomic Computing.

2.2 Monitoring and Reconfiguration Operations

Run-time monitoring and reconfiguration are essential operations to enable SAA systems. Monitoring operations perform diagnosis and state tracing of systems and circuits via physical (e.g., thermal) and logic (e.g., cycle counter) sensors. Reconfiguration is the operation to adjust the systems based on the collected information in the monitoring process. A large diversity of monitoring and reconfiguration operations have been proposed on parallel embedded systems (Table 2.1), with the main purposes to improve system performance, energy efficiency, and dependability.

Performance is a conventional concern of monitoring and reconfiguration operations. Major components of a parallel embedded system, including processing elements, networks, memory and I/O, are all targets of performance monitoring and reconfiguration. For instance, cache misses and CPI (cycle-per-instruction) can be traced for each processing element [125] to monitor its performance and identify possible causes. Reconfiguration can be issued based on the monitored information to improve the performance. For example, the schedulers can reduce the simultaneous execution of memory-bound tasks when the processor-memory interconnection becomes heavily loaded [125].

In addition to performance, reconfiguration towards energy-efficiency is also applied in many works [80, 37]. There are techniques addressing different power or energy metrics, including dynamic power [71], dynamic energy consumption [37], and static power [80]. The run-time power or energy optimization is usually performed as a tradeoff with performance or dependabil-

Table 2.1: The Diversity of Monitoring and Reconfiguration Operations in Parallel Embedded Systems

Targets	Explanation	Examples	Monitored Parameters
Function	To support new functions (application-specific)	Monitoring soil humidity to adjust watering in an agricultural scenario [23]	soil humidity
Performance	To achieve higher speedup, resolve contention	Adaptive routing to reduce network congestion; Dynamic buffer allocation for intensive traffic flow [84]; Dynamic resource allocation (suggested in [108])	network load, buffer occupancy ratio, processing speed
Hard Failures	To detect and recover the system from permanent errors	Using redundant cores to replace failed processors [105]; Using redundant wires to replace broken links [68]	CPU failure; interconnect error
Temporary Errors	To detect and recover system from transient errors	Dynamic retransmission for transient faults in communication; Error detection and correction in processor and links using Razor architecture [27]	Run-time communication data; Flip-flop content (Razor)
Power/ Energy	To reduce power/energy consumption or provide energy-performance tradeoff	Adaptive power reduction with dynamical voltage and frequency scaling [37]; Dynamic power gating to reduce static power [80]	current (either in active or idle states); timing slack
Temperature	To avoid temperature hotspot or thermal breakdown	Run-time sensing of thermal hotspot and dynamic voltage scaling to avoid thermal breakdown[106]	temperature (thermal sensors using, e.g., current)

ity. Thus the monitoring is instrumented via tracing the activity profiles (e.g., buffer occupancy and link utilization [110]) and the error profile [126]. The system is reconfigured to reduce the power or energy consumption while meeting the performance or dependability constraints (e.g., error rate [126]). Common reconfiguration techniques include Dynamic Voltage and Frequency Scaling (DVFS) and its many variations, dynamic buffer allocation [63], clock/power gating and dynamic mapping [2].

Besides, monitoring and reconfiguration to provide dependability are also much needed operations [68, 105]. Dependability is a broad concept, covering the notions of availability, reliability, safety, integrity and maintainability [6]. In practice, fault tolerance is a major design concern for high-performance embedded computing [105, 27]. A diversity of monitoring interfaces have been proposed, including error detection coding, specific counters to identify performance degradation [56] and inline testing [68]. With the awareness of permanent and transient errors, proper reconfiguration can be performed, such as ARQ (automatic repeat query) for transient faults, and spare wire replacement for permanent errors [68].

From these existing works on run-time monitoring and reconfiguration, we can firstly observe that such techniques are so widely applied to different types of components (processors, networks, memory, etc.) that dedicated design to integrate proper monitoring and reconfiguration services becomes essential for modern embedded systems. Secondly, the techniques addressing performance, energy efficiency and dependability are often related to each other. For instance, Razor II architecture [27] employs aggressive voltage scaling on the processor to minimize the energy consumption, while tracing the soft errors caused by voltage scaling and PVT variations. While error-observing voltage scaling is applied to processors in [27], [126] performs error transmission rate based Dynamic Voltage Scaling (DVS) for energy-efficient on-chip interconnects. In order to integrate the large diversity of monitoring and reconfiguration operations in a real-life complex system, systematic approaches that can be utilized on various platforms need to be developed.

2.3 Paradigm Evolution towards 3-Dimensional Design Space

The development of SAA embedded systems is a highly complex process, with many considerations in the design of computation, communication and control logic. To reduce the development efforts and maintain an affordable time-to-market, two system-level approaches need to be applied: design orthogonalization to narrow down the design exploration, and Platform-Based Design (PBD) to reuse existing architectures.

The orthogonalization of design concerns [58] is one major system-level principle to reduce the design complexity. The focus of study on specific concerns does not exclude the interactions with other aspects, but rather motivates the concentration of design efforts to improve the efficiency of the system development. Current design space exploration mainly consists of two dimensions: data processing and communication. The studies of data processing address the data parallelization, processing on individual elements and the performance of overall parallel systems. The studies of communication involve the analysis of communication architectures (e.g., message passing or shared memory space), specific implementation of communication channels (such as physical and virtual channels), as well as functional and non-functional optimization techniques on these architectures, e.g., power optimization.

PBD [100, 58] is a modern paradigm, which originates from the classic concept of Model-Based Design (MBD) [104, 53]. MBD abstracts the components (processing elements, communication components or control logics) into models with functional, timing and communication specification. It was the initial design revolution towards efficient design reuse and easier verification at an early design stage. With numerous types of hardware components and software libraries to be integrated, constructions from individual models may not be efficient for a highly complex system. Instead, PBD, which groups components at a specific level into a development platform, is widely adopted for system development. The definitions of platforms vary depending on the application domain. Conceptually we can perceive the platform as an abstraction layer that captures the functional and non-functional features of a particular design level. PBD is intrinsically hierarchical since each architectural level is characterized by a specific platform. Compared to MBD, it improves the design efficiency by allowing the reuse of a platform at a high level of abstrac-

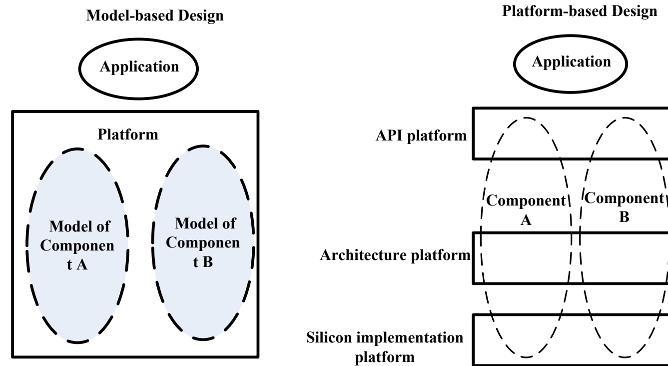


Figure 2.3: Conventional Model-based Design vs. Platform-based Design

tion for various applications. In the meantime, PBD still supports flexible implementation alternatives, especially by exploiting reconfigurable technologies to modify the details of the platform and tailor it to different application requirements. Common platform stacks in PBD [101] and its comparison with MBD are illustrated in Fig. 2.3.

The design principles of orthogonalization and PBD have motivated the design evolution of communication-centric design (Fig. 2.4). Previously, the design was focused on data processing, while the communication was dealt with in an application-specific manner, either using buses or point-to-point connection. Communication-centric design, in particular NoC, separates the functional behaviors of cores (processing) and the interaction between the cores (communication) [93], in order to improve design efficiency and provide a generic communication architecture for reuse.

As monitoring and reconfiguration operations become instrumental to the realization of SAA systems, the design of these operations deserves focused attention. Currently, these operations are designed in an application-specific or ad-hoc manner, which prevents efficient design reuse. In fact, as discussed in Section 2.2, various monitoring and reconfiguration techniques can be applied to different platforms. To effectively design SAA systems with minimized design efforts, we propose a dedicated design dimension for adaptivity, leading to a 3-D design space for parallel embedded system (Fig. 2.4). The separation of a design layer enables the designers to focus on building a portable and scalable architecture for SAA properties, which can be added upon existing data processing and communication infrastructure (design reuse). The thesis

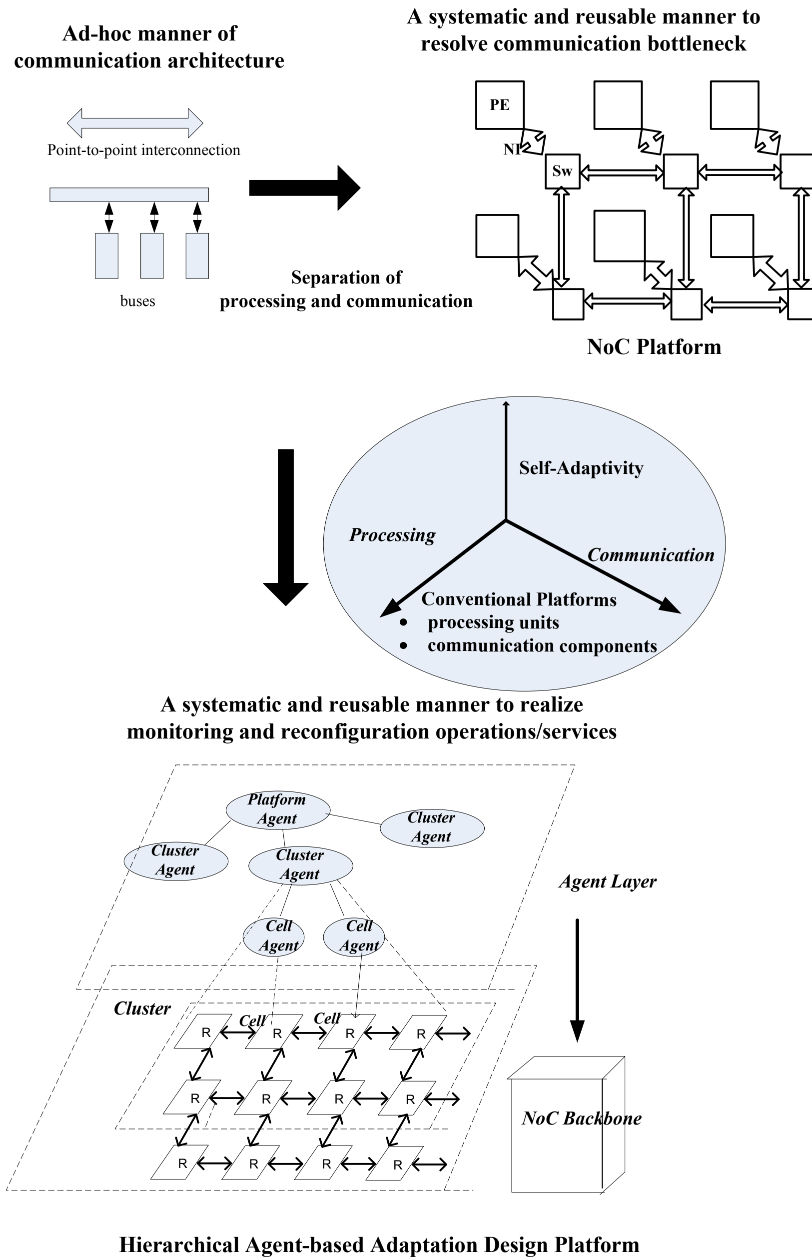


Figure 2.4: SAA as a Separate Design Dimension

is dedicated to the design and realization of such a design layer.

2.4 Chapter Summary

This chapter motivates SAA systems in parallel embedded computing, and reasons the necessity of a dedicated design layer for SAA properties. To deal with the increasing complexity under hardware variations and diverse application scenarios, a self-managing system is more capable of achieving run-time optimization compared to worst-case-based design. This chapter presents an abstract model of the SAA system, and explains its difference from autonomic and reconfigurable computing. Then it overviews the monitoring and reconfiguration services in modern embedded systems, which are utilized to realize SAA properties. Following the principle of design orthogonalization and PBD, the chapter proposes a separate design dimension addressing monitoring and reconfiguration operations for self-adaptation services. Adding such a design layer will achieve a reusable platform where SAA properties can be designed upon existing processing and communication architectures.

Chapter 3

Design Exploration of Self-Adaptation Architectures

While self-aware and adaptive systems may be realized in various architectures with different functions and objectives, the self-adaptation procedure is fundamentally composed of three distinct but related processes: monitoring (M), decision-making (D) and reconfiguration (R). Most existing works address novel techniques in specific M, D and R processes [29, 62]. In particular, for many-core systems, energy management has been one primary target of self-adaptation, where the tradeoff between power/energy and performance can be made at the run-time [64]. What misses in the previous efforts is a study of the integration of all M, D and R processes into a self-adaptation architecture, and a comparative study of diversely integrated architectures. In addition, current physical design may have specific support for run-time reconfiguration, for example, on-chip voltage regulators. Thus, this chapter presents a comparative evaluation of various self-adaptation architectures for many-core systems. In particular, centralized, clustered and distributed techniques are differentiated to categorize these architectures. These architectures are simulated on a NoC simulator for the examination of the communication energy, latency and energy-delay product, with the network congestion as an exemplified performance metric under monitoring. Based on the comparative studies, we also identify the need for hybrid management architectures to enable wider utilization of self-adaptation techniques.

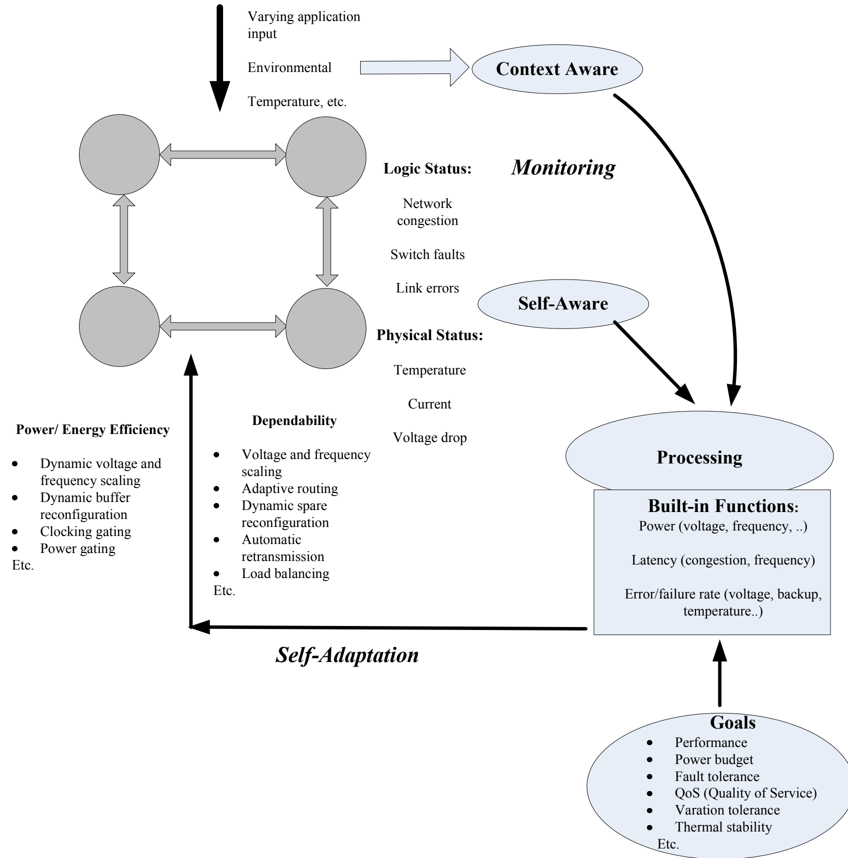


Figure 3.1: Monitoring, Decision-Making and Reconfiguration Processes in Self-Adaptation Architecture

3.1 Monitoring, Decision-Making and Reconfiguration Processes

Monitoring (M), decision-making (D) and reconfiguration (R) form the closed-loop self-adaptation procedure (Fig. 3.1). Monitoring ensures the system visibility of its own status and performance, which is the prerequisite for justified reconfiguration addressing the actual system needs. Decision-making is the analysis process, where a controller decides on the proper reconfiguration to be performed, given the built-in objectives or goals. Reconfiguration is the changing process of system parameters via actuators, for instance, the updating of a multiplexer. In a parallel system, when the monitors, decision-makers and actuators are distributed, monitoring communication is needed between them.

3.1.1 Monitoring

Monitoring, or the tracing and sampling of system and circuit status is a widely studied process. In particular, distributed monitoring in parallel embedded systems has become highly feasible, efficient and low-cost in modern technology [29, 56]. Before the wide emergence of multi-core or many-core architectures, the monitoring of single processor performance (e.g., the memory access) using dedicated hardware with affordable overhead, has already been common practices [111]. In the many-core system era, different processor architectures are integrated as IP blocks into parallel systems, thus the distributed monitoring in each processor is feasible as an internal structure of each IP block. The distributed monitoring of network components, for instance, routers or links, is also technically feasible. For instance, distributed probes are added to network interfaces in a NoC[29], which can monitor information such as throughput, timing or latency. The area overhead is only 27% of a network interface's area. In addition, fault detection can also be provided by distributed monitoring. For instance, a hop counter can be added to the packet [56], which increments by one when passing a new switch. If the counter overflows, a routing error may have occurred, as the packet goes through on a much longer path than it should (assuming the routing algorithm is livelock free).

3.1.2 Decision-Making

With the monitored information, the decision-making process determines the reconfiguration operations based on specific cost functions. Common cost functions include performance, power consumption and dependability. The decision-maker can be centralized or distributed (Fig. 3.2), with trade-offs in scalability, local and global optimization as well as overheads.

In case of centralized decision-making, the overhead is small since only one decision-maker is required for the whole system. In particular, a software-based decision-maker is able to support extensive, flexible and reconfigurable functions. ElastIC [113] proposes an architecture composed of many tunable processing elements and one centralized DAP (diagnostic and adaptivity processing) unit. Each tunable processing element needs to have the interfaces for run-time monitoring and reconfiguration, while the DAP unit dynamically performs diagnosis and reconfiguration for the processing elements. Central-

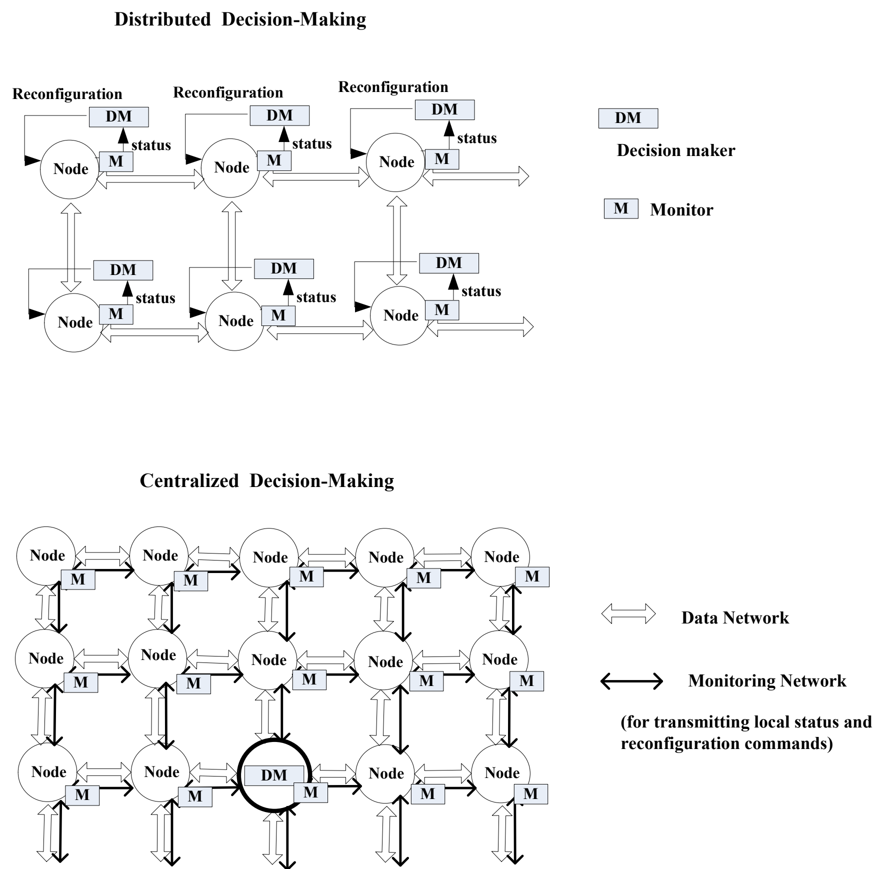


Figure 3.2: Illustrating Centralized and Distributed Decision-Making on NoCs

ized decision-making suffers from non-scalable burden on the decision-maker as the system expands. Firstly, the control module will become highly complex if the decision-maker needs to respond to many requests from all components. Besides, the communication from every component to a centralized decision-maker can easily create communication congestion (Section 3.2.2). Also there will be energy overhead for such communication.

In case of distributed decision-making, if each controller is a general purpose processor (i.e., a software-based decision-maker), the system overhead is very large. Using individual threads for decision-making is much more cost-effective (to be elaborated in Section 7.3. If the decision-making function is simple, hardware-based controllers are suitable with low overhead. For instance, [115] presented hardware-based controllers for per-core DVFS in each processor on a 167-core computing platform. The controller is very small, only covering 3% of each processor area. With distributed decision-makers, the monitoring communication overhead is minimized as the system status and reconfiguration commands are transmitted locally (Fig. 3.2). However, the distributed controllers do not have the global information, thus are unable to target global optimization. In contrast, centralized controllers have the global view to optimize the overall system performance. For instance, in a chip with 2 processors and two media-processing hardware IPs [98], a small processor core (centralized) controls the chip temperature while meeting the real-time constraints. As the computation may be performed on more than one processing element, the centralized controller can decide on the proper reconfiguration based on its awareness of the status of all involved components.

3.1.3 Reconfiguration

After the decision-making process, reconfiguration can be performed either uniformly to the whole system (centralized reconfiguration) or individually to different components (distributed reconfiguration). For power and energy management or dependability, there exist a wide range of reconfiguration techniques [39] such as voltage and frequency switching, dynamic buffer allocation, clock and power gating, and dynamic mapping. Our discussion omits technology-dependent mechanisms, for instance, the partial reconfiguration in FPGAs.

In terms of local optimization, the distributed reconfiguration is more effective than the centralized reconfiguration, as the workload or traffic spatiality

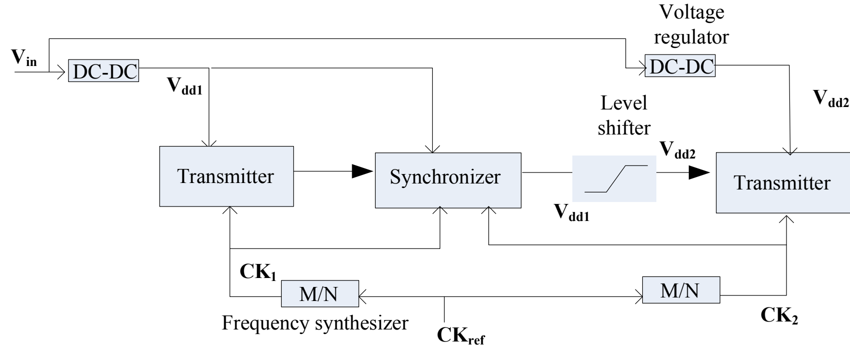


Figure 3.3: Overheads of Fine-grained Reconfiguration for Individual Voltage and Frequency Domains

can be better accommodated with individual reconfiguration. For instance, it is shown in [62] that per-core DVFS achieves up to 21% energy saving compared to conventional centralized DVFS applied uniformly to the whole chip.

However, the realization of distributed reconfiguration is more challenging than centralized reconfiguration, particularly due to the synchronization issues and reconfiguration overheads (Fig. 3.3). The synchronization between two frequency domains requires synchronizers to minimize metastability. Level shifters are needed to interface two different voltage domains. Both synchronizers and level shifters incur time, area and energy overheads [81, 33]. For instance, a bi-synchronous FIFO [81] can be added between two asynchronous routers. The minimal FIFO depth to provide 50% throughput is 5, which clearly incurs area and energy overhead. In addition, the timing overhead of the FIFO structure is between two and three reading clock cycles. In addition, the voltage regulator for dynamic voltage scaling, despite the improvement in its conversion efficiency, still pose considerable area and power overhead [45].

3.1.4 Monitoring Communication

Interconnection is needed to support the communication between monitors, decision-makers and actuators. We generalize such networking as the monitoring communication. Specific properties are required for such communication. Firstly, it has to be provided with guaranteed service. In particular, the flow should be isolated from the application data communication. Otherwise, if the network is unpredictably congested by the application data, the monitoring information will be lost. Second, the latency and energy overheads of

the monitoring communication should be minimized, if not with strict boundaries. Several alternative architectures can be applicable, including physically separate network [76], TDM (time-division multiplexing) [36] and CDM (code-division multiplexing). The simple architecture of physically separate networks reduces the design and verification efforts, which is desirable in modern parallel embedded systems. In contrast to intuitive assumptions, physically separate networks are highly feasible in current technology, as multi-layer fabrication provides abundant wiring resources on-chip [124]. For instance, TILE64 multiprocessor integrates 5 physically separate networks, each only accounting for 1.1% of the die area [124]. Other more complex communication architectures may require significant design effort and incur energy overhead. For instance, TDM-based monitoring network consumes much more energy than physically separate monitoring networks based on [36]. Considering that the future systems are strongly energy-limited, in this chapter a physically separate network for the monitoring communication is assumed (Fig. 3.2). Section 6.3 will further explore this issue.

3.2 Centralized, Clustered and Distributed Architectures

Run-time self-management architectures need to integrate the M,D and R processes. Modern VLSI technology has been giving physical and circuit-level support to enable run-time monitoring and power reconfiguration on many-core systems. Based on these supports, we can design coarse- and fine-grained self-adaptation architectures.

3.2.1 Physical Support for Voltage and Frequency Reconfiguration

Voltage regulators (VR), either on-chip or off-chip, are the classical techniques to enable voltage reconfiguration (Fig. 3.4). Conventional off-chip regulators are usually slow, due to the large parasitics [34]. To increase the speed of voltage scaling, on-chip voltage regulators are proposed, which achieve frequencies on the scale of $100MHz$ [62]. Regardless of the improvement on speed and energy efficiency, VR-based approaches incur noticeable area overhead. For instance, [102] presents a highly-efficient on-chip voltage regulator in $65nm$

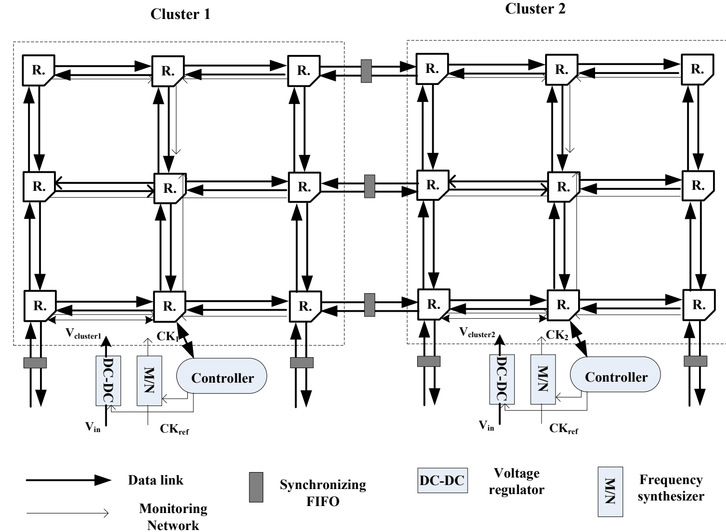


Figure 3.4: Voltage Regulator based Energy Management

CMOS consuming $0.77mm^2$ area. If each processor on a 80-core chip required such a regulator, the total area overhead for voltage switching would amount to $61.6mm^2$ on a $275mm^2$ chip [119].

Considering the overhead of voltage regulators, a new technique, multiple on-chip power delivery networks (MPN), provides a fast and lower-cost approach (Fig. 3.5 [115]). Several global power delivery networks are implemented on high metal layers. The local power networks inside each component are routed on middle metal layers. The voltage of each component can be dynamically connected to one of the global power lines via power switches. As demonstrated by [115], the voltage scaling from 0.9V to 1.3V takes less than $4ns$ with clock halting, which is significantly faster than voltage regulators. Power switches are implemented as parallel transistors, whose area overhead is analyzed as only 4% of a node area [115]. However, the MPN-based platform is demanding in the physical design process, including the layout of multiple power delivery networks and the design of fine-grained power switches to provide low voltage-drop and dependable voltage transition. Thus this technology has not been widely adopted except in few works such as [115, 14].

Compared to voltage regulators, the frequency synthesizers used for runtime frequency scaling incur much lower overhead in time and area. Even with $90nm$ technology, the clock generator in [89] only covers $0.05mm^2$ area with $4.5ns$ switching time. As the frequency switching delay is overlapped with the

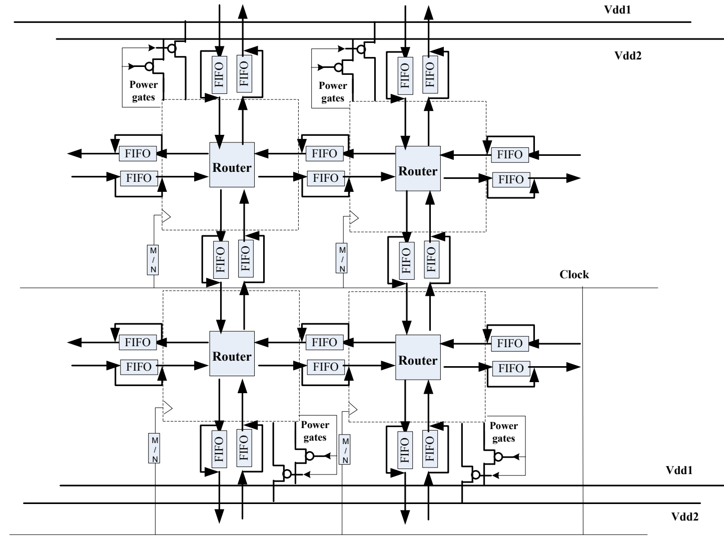


Figure 3.5: Multiple On-Chip Power Delivery Network based Per-Core Voltage Switching

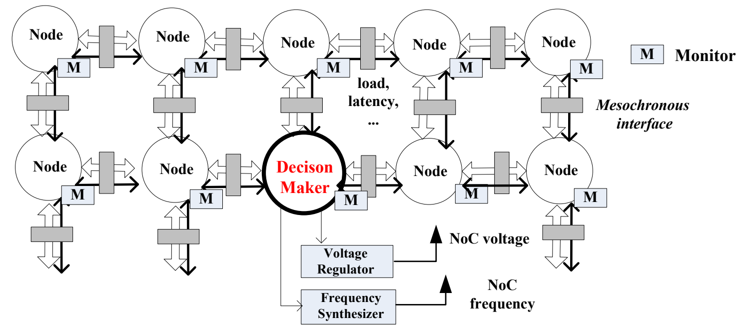


Figure 3.6: Centralized Energy-Management Architecture

voltage regulation time, no additional timing penalty is incurred.

3.2.2 Centralized Architecture

In conventional energy management architecture, a centralized decision-maker [71] supervises the monitoring and reconfiguration for the whole system (Fig. 3.6), while the run-time information is still gathered locally. The decision-maker reconfigures the network (e.g., its voltage and frequency) in a unified manner. Despite that the network runs at the same frequency, it is difficult to ensure that all clocks in every router keep the same phase, thus mesochronous interface is needed [119].

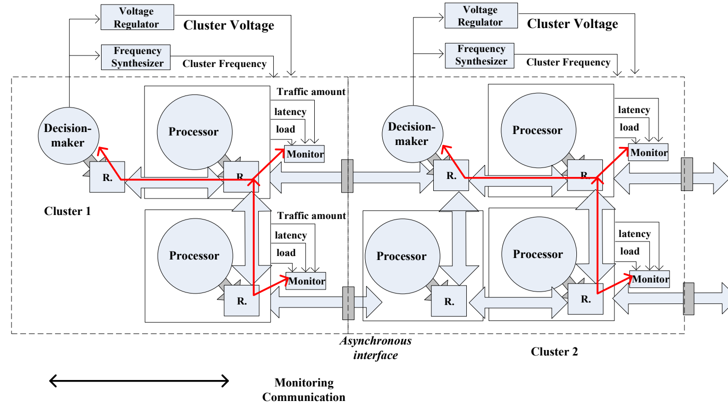


Figure 3.7: Cluster-based Decision-Making and Reconfiguration

The centralized architecture has low area overhead, since only one set of voltage regulator and frequency synthesizer are needed. However, chip-wide reconfiguration cannot address the spatial locality of the workloads, as will be demonstrated in Section 3.3. It is only suitable for networks that have uniform traffics across the system. In addition, as all nodes send monitoring information to a single decision-maker, congestion may appear when there are a large number of nodes.

3.2.3 Clustered Architecture

As the centralized architecture cannot account for the workload's locality, cluster-based energy-management has been proposed [90, 129]. Instead of a centralized decision-maker, the system is divided into several clusters, each of which being supervised by a decision-maker (Fig. 3.7). The reconfiguration is applied to the whole cluster. Between the clusters, asynchronous interface is needed as frequencies can be different at the two ends.

The overhead of VR-based clustered architecture, due to the voltage regulators, is higher than that of the centralized architecture. But the total number of regulators only grows with the number of clusters, not with the number of processors. For instance, with the voltage regulator as in [102] ($0.77mm^2$), a system with four clusters will consume $3.08mm^2$, as 1.12% of a $275mm^2$ chip. The major benefit of this architecture lies in its adaptation to regional traffic loads, thus it is suitable for systems with multiple applications. However, the voltage regulator still incurs considerable timing overhead (in the range of $100ns$ including the voltage stabilizing time on the power line [62]). In case

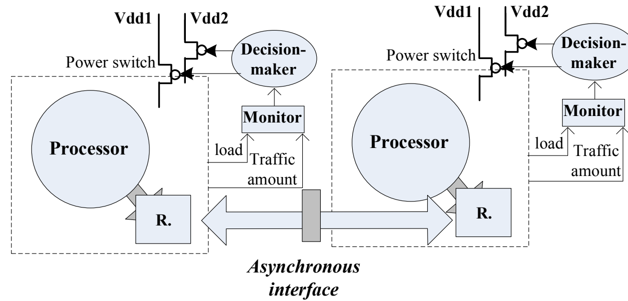


Figure 3.8: Distributed Energy Management with MPN-based Physical Support

of fast-changing on-chip communication, such slow voltage switching may prevent the network from adapting to the traffic variation on time, resulting in low energy efficiency (Section 3.3 will demonstrate this issue). In addition, the workload variations within a cluster cannot be addressed when all nodes in the cluster receive the same reconfiguration.

3.2.4 Distributed Architecture

In the distributed architecture, each local decision-maker supervises the reconfiguration of the corresponding node, as illustrated by Fig. 3.8. Thus, there is no global networking between the monitors and the decision-makers. To enable such fine-grained reconfiguration, especially for voltage switching, the multiple voltage delivering networks with power switches are necessary. As discussed in Section 3.2, the parallel-transistor-based power switch only requires 4% of each network node's area [115], thus significantly smaller than distributed voltage regulators. Since each router may run on a different frequency and voltage, asynchronous interface (e.g., with FIFOs and level shifters) is needed on each link.

Compared to the clustered architecture, distributed energy-management can respond to even more fine-grained workload locality, as each router can run on a different frequency. More importantly, as MPN-based voltage switching is significantly faster than voltage regulators, the energy-management can capture the fast changes in the traffics, leading to superior energy-performance tradeoff. However, the asynchronous interface on each link incurs noticeable energy and timing overheads. In addition, local adaptation may overact to temporary workload variations, which may lead to oscillation with worse per-

formance.

3.3 Quantitative Comparison

Here extensive quantitative evaluation is performed to compare the effectiveness and efficiency of various self-adaptation architectures. In particular, the energy consumption, performance and energy-performance tradeoff in NoC communication will be analyzed. The influences of switching delay, synchronization overhead and monitoring communication will be identified and compared. A set of representative synthetic traces are utilized to provide a generally-applicable conclusion. The quantitative study in this chapter excludes area overhead comparison, as it becomes a secondary design concern with constant technology scaling. Besides, previous works [37, 115] have shown the feasibility of clustered and distributed energy management in NoCs.

3.3.1 Platform Setting

A flit-level accurate and trace-driven NoC simulator is built for simulating monitoring and reconfiguration techniques (see Appendix A).Each tile in the NoC is a $2mm \times 2mm$ square. In case of centralized and clustered architectures, there are two routers in each tile for the data and monitoring communication respectively. The distributed architecture only has routers for the data communication. The data router is input buffered with 2-flit input buffers. The router utilizes wormhole-based switching and X-Y deterministic routing. Every packet is in the form of one header flit followed by 7 payload flits. Each flit is 32-bit wide, so each channel has 32 bits per direction. The monitoring network uses a similar router architecture with one-flit-deep input buffers and 8-bit channel width, as its communication volume is typically lower than that of the data traffic.

Based on the analysis of a bi-synchronous FIFO architecture [81], a 5-flit FIFO is assigned on each asynchronous interface with 3 reading cycle latencies between different frequency domains, while a 4-flit FIFO is used for each mesochronous interface with 2 reading cycle latencies. The delay on the level shifter is negligible compared to the FIFO delay [33], thus omitted. During the voltage switching, the involved nodes are paused. For VR-based switching, the delay is set as $100ns$ (Section 3.2.3). For the MPN-based switching,

the delay is much lower as $10ns$ with clock gating (realistic based on [115] including the voltage stabilizing time on the power line).

Energy is modeled by calculating the consumption of each packet traversing the routers and links. The choice of voltage and frequency pairs is dependent on the implementation, for instance, the critical path. In this work, we adopt the values from [14] for experimental purposes, as the detailed circuit-level design is beyond the interests of our architectural comparison. Two pairs of voltage and frequency values are (V_H=2V,F_H=1GHz),(V_L=1.05V,F_H=$\frac{1}{3}$GHz). As SoC approaches lower voltages, designing more than two discrete voltages returns little benefits [19] while the overhead of power network distribution increases. Given the voltage and frequency, the energy consumption of routers and links can be obtained from Orion 2.0 tool [55]. We estimate the energy consumption of synchronization interfaces also with Orion 2.0, since the FIFOs are usually designed with the same shift register structure as the input buffers of the router. In terms of energy overhead of voltage regulators, it follows $E = C \times (1 - \mu) \times |V_{dd2}^2 - V_{dd1}^2|$ [112]. C is the filter capacitance of the power-supply regulator. μ is the conversion efficiency. V_{dd2} and V_{dd1} are the voltages before and after the transition. In the experiments, C is configured as $2.5nF$ and μ as 82.5% [45]. Th energy consumption of the level shifter is from the figure in [33].

3.3.2 Traffic Generation

In order to evaluate the energy-performance tradeoff for a general-purpose many-core platform, a set of synthetic traces with distinctive temporal and spatial variation patterns are devised and simulated. The synthetic traffic traces are categorized by two features- the temporal injection rate and the destination locality:

- In terms of injection temporal rates from each processing element, we consider uniform and b-model traffics [123]. Uniform pattern represents an ideal case of network traffic. To accommodate traffic variations in the experiments, we generate b-model traffics [123]. It models self-similar and temporally varying traffics, which are demonstrated to be realistic in practical network environments [123] and utilized in on-chip communication microbenchmarks [74]. A single parameter $b(0.5 \leq b < 1)$ can be set to configure the burstiness of the injection. The closer b is to 1,

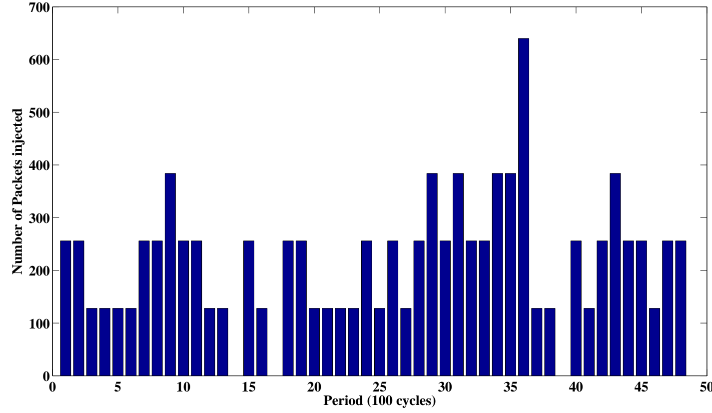


Figure 3.9: An Example of B-model Traffic (b=0.6)

the more bursty the traffic becomes. Fig. 3.9 illustrates a b-model traffic trace with $b = 0.6$.

- In terms of packet destination distribution, we consider uniform and hotspot traffic. Uniform traffic assumes the same probability of destination for all packets. Hotspot pattern assigns a high transmission probability to certain regions of the network. Such pattern of communication is likely to appear when certain processors are major data consumers in a parallel computing platform. Eq. 3.1 and Eq. 3.2 model the probabilities of a node in the hotspot region and other region as the transmission destination respectively. ρ is the fraction of the traffic targeted to the hotspot region. $N_{hotspot}$ is the number of nodes in the hotspot region. $N_{network}$ is the total number of nodes in the network.

$$P(hotspot) = \rho / N_{hotspot} \tag{3.1}$$

$$P(other) = (1 - \rho)/(N_{network} - N_{hotspot}) \tag{3.2}$$

The combination of temporal and spatial patterns results in four traffic traces, which run simultaneously on an 8x8 mesh-based NoC, as illustrated by Fig. 3.10. The traffic pattern is labelled as a pair listing the temporal and spatial pattern. For instance, (U, U) stands for uniform temporal injection rate and uniform destination distribution. The temporal injection rate (R),

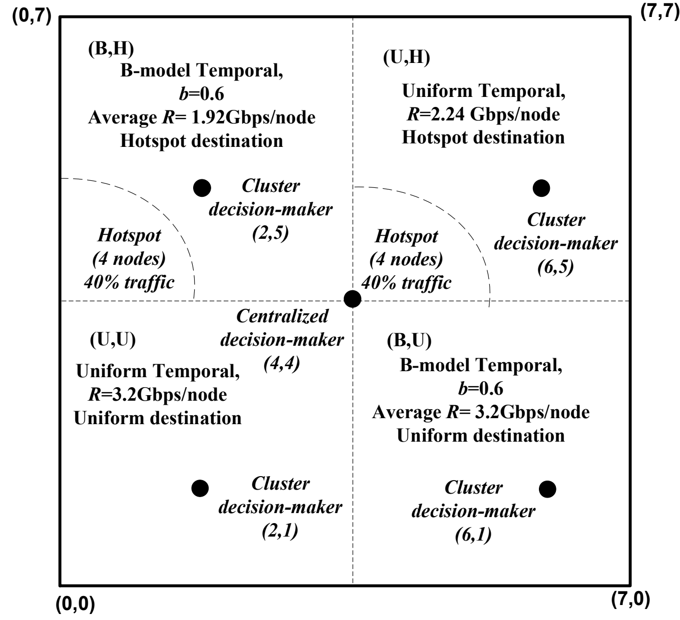


Figure 3.10: Experimental Platform Running Four Traffic Traces (each trace runs in a 4*4 cluster)

hotspot location, bursty parameter b, and decision-maker locations are also labeled with experimental values.

3.3.3 Simulation Results

The energy management, in particular energy-performance tradeoff, with different architectures is simulated with DVFS adapting to the network traffic. Buffer load, the percentage of buffers occupied in routers and synchronizing FIFOs of the interested area (individual router, a cluster, or the whole network), indicates the traffic congestion. The higher the buffer load is, the longer waiting time on average the packets have. Thus the decision-makers adjust the frequency and voltage of the corresponding routers, in order to keep the buffer load within a design-specific range. For the distributed architecture (DI), per-router DVFS is performed, where each local decision-maker adjusts the voltage and frequency based on the buffer load of the corresponding router. For the clustered architecture (CL), per-cluster DVFS is performed, where the buffer load is an average from all routers in the cluster. For the centralized architecture (CE), the buffer load is an average of all routers in the NoC. In our simulation, the design-specific buffer load range is chosen as $(0.1, 0.2)$. If

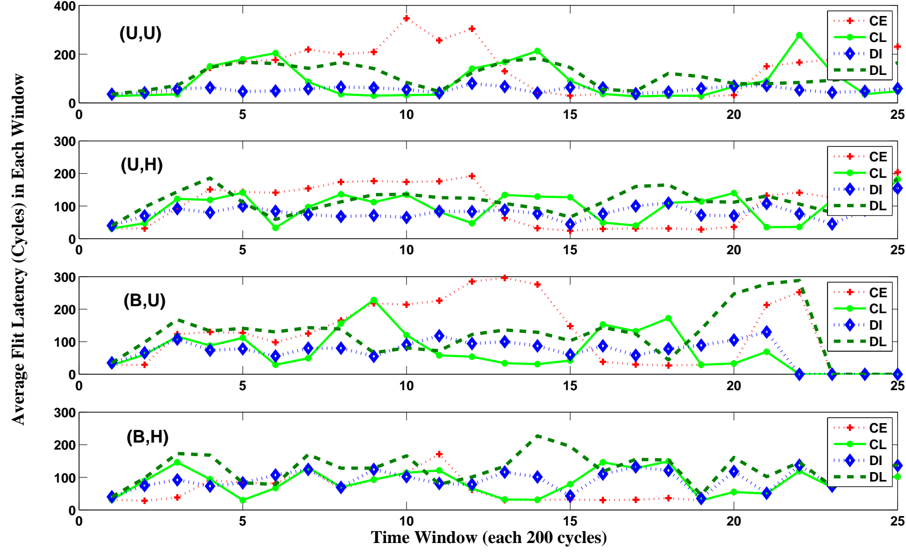


Figure 3.11: Temporal Latency of Four Traffic Traces in Different Energy Management Architectures

the buffer load is above 0.2, the corresponding routers are reconfigured with the high frequency and voltage. Likewise, the routers are reconfigured to the low frequency and voltage in case the load is below 0.1. The report of buffer load in the monitoring communication, in case of the clustered or centralized architecture, contains 2 flits. The buffer load is always, on the timescale, an average of figures in each time window of 200 cycles.

3.3.4 Performance

The average flit latencies of the four traces running in the three architectures (*CE*-centralized, *CL*- clustered, *DI*- distributed) are depicted in Fig. 3.11. The latency is reported in every time window (200 cycles at $1GHz$). For analysis purposes, an additional setting (*DL*) with distributed DVFS with a long switching delay ($100ns$, the same as the voltage regulator) is also simulated to identify the influence of the switching delay.

It can be observed from Fig. 3.11 that there are clear performance differences with different energy management architectures. The average latency of the centralized architecture (*CE*) drastically changes in different periods. It is caused by the centralized reconfiguration that does not consider the run-

Table 3.1: Average Flit Latency (Cycles) in Monitoring Network

CE	Trace (U,U) in *CL*	Trace (U,H) in *CL*	Trace (B,U) in *CL*	Trace (B,H) in *CL*
558	64	84	70	84

time network load of each application. The clustered architecture (*CL*) has significantly lower and more stable latency compared to *CE*. The distributed architecture (*DI*) has the lowest and the most stable performance.

The performance differences under different energy management architectures are caused by four factors. Firstly, the granularity of reconfiguration process determines if the system can adapt to traffic's spatial locality. Thus, the finest-grained distributed architecture has the lowest latency. Second, the switching delay affects if the network can speedily complete the reconfiguration. This issue can be further elaborated through the experiment of distributed DVFS with a long switching delay (*DL*). As can be observed from Fig. 3.11, when the delay is configured as long as the voltage regulator, the latency of the distributed architecture is significantly increased to be comparable with *CL*. Third, the congestion in the monitoring network is an issue for the centralized architecture, as can be seen from Table 3.1. Last but not least, if the network switches too often to induce oscillation, the performance is negatively influenced. From Fig. 3.12, it can be seen that the distributed architecture suffers from the largest number of switching. Frequent switching incurs performance penalty, which is clearly seen when the delay is configured longer in the case of *DL*.

3.3.5 Energy Efficiency

The average communication energy consumption of all four traces with different architectures is reported in Fig. 3.13, labeling also the overheads of synchronization FIFOs and monitoring communication. It can be observed from Fig. 3.13 that the energy consumption of the *CE*,*CL* and *DI* is comparable with no convergent winners in the four traffic traces. It is not a surprise that the energy consumption of distributed DVFS is not necessarily the lowest, since the other two architectures may reduce more energy by sacrificing the performance. The synchronization FIFOs add a small but steady share of

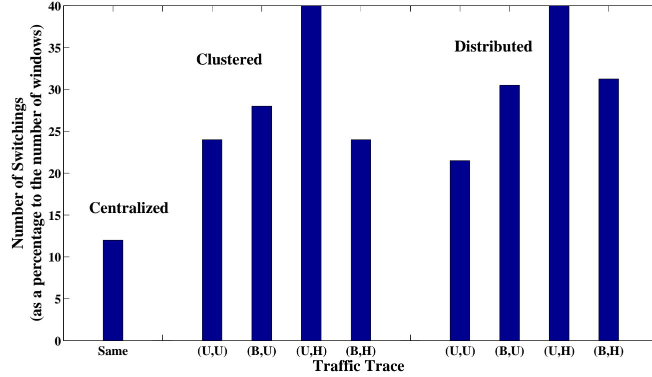


Figure 3.12: Number of Switchings in Different Energy Management Architectures

energy consumption (around 2.5%). Besides, the monitoring communication also adds small energy overhead (maximal 4.8% for the centralized architecture, 4.6% for the clustered architecture), due to its low volume compared to the data communication. In addition, the relative energy overhead from the voltage regulator in clustered and centralized architectures is very small (less than 1%), thus omitted. As the data communication volume steadily increases in NoCs, the relative energy overhead from the voltage switching will become insignificant given a small number of regulators.

As the latency values of the three architectures differ significantly, to compare the energy efficiency, Energy-Delay Product (EDP) can be analyzed (Fig. 3.14). In this context, the delay is interpreted as the communication latency.

From Fig. 3.14, it can be observed that the clustered architecture is 12.1% and 43.9% lower in EDP for (U,U) and (B,U) traffics compared to the centralized architecture. The per-core DVFS has a more clear advantage. Its maximal saving is 50.7% compared to the centralized architecture, and 43.9% compared to the clustered architecture. The only counter-example is the (B,H) traffic. Since the centralized architecture utilizes extra energy for this traffic based on the average buffer load in the whole network, (B,H) traffic alone gets good performance while the other three traces suffer from long and drastically changing latencies. The analysis on EDP further demonstrates the energy efficiency of clustered and distributed management in NoCs.

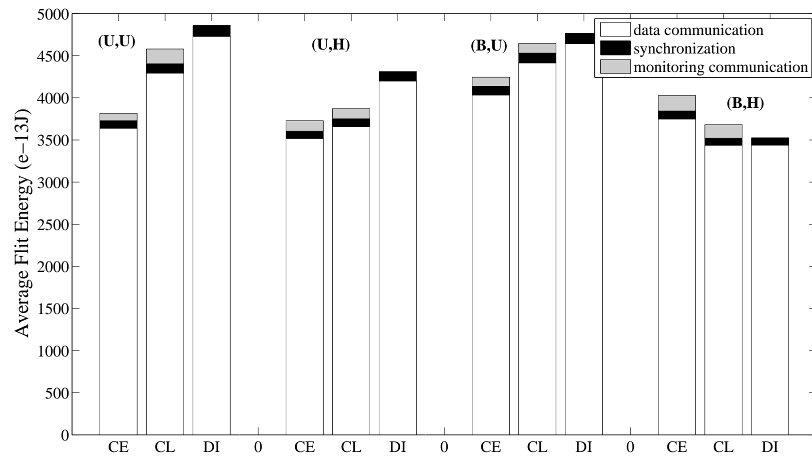


Figure 3.13: Average Communication Energy of Four Traces in Three Energy Management Architectures

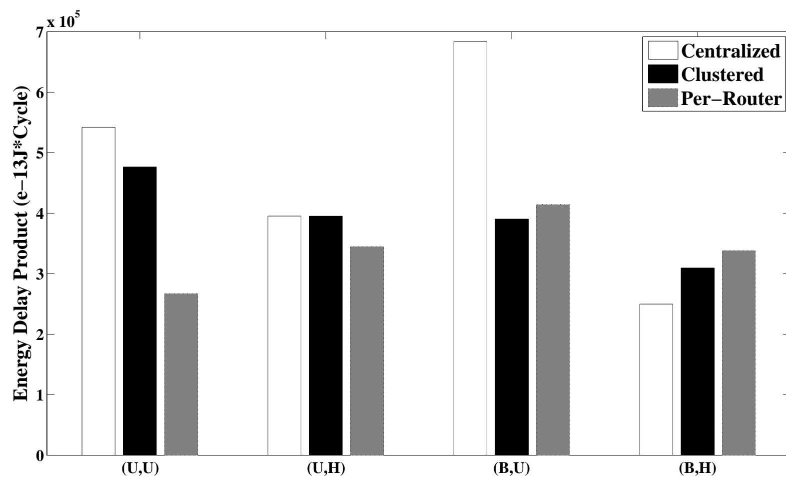


Figure 3.14: Energy-Delay Product of Four Traces with Different Energy Management Architectures (*CE*, *CL* and *DI*)

3.4 Summarizing Comparison

Based on the qualitative and quantitative comparison in Section 3.2 and 3.3, the pros and cons of the centralized, clustered and distributed architectures can be summarized as in Table 3.2.

The centralized architecture, though with the most simple design, lacks in adaptation to spatial locality. The monitoring communication also leads to congestion in this architecture. The clustered architecture, with much better scalability, is enabled with high-speed voltage regulators, and significantly improves the energy-performance tradeoff. However, the switching delay of voltage regulators is still noticeable for fast-changing on-chip traffics. Thus the distributed architecture with MPN is proposed. The simulation result shows that the energy efficiency of the distributed DVFS is higher or similar to the clustered DVFS, while the performance is more stable with lower maximal temporal latency. Such advantage is due to that distributed DVFS has more fine-grained local adaptation with faster switching frequency. However, none of the architectures directly addresses performance metrics, for instance, latency. Hence the communication performance varies with time and traffic patterns.

3.5 Chapter Summary

This chapter studies and compares the design of adaptive services in parallel embedded systems evolving from the centralized architecture towards the distributed architecture. The study exemplifies the adaptive energy management service in NoCs, in particular the energy-performance tradeoff. The comparison shows that the centralized architecture is in general non-scalable, when addressing localized parameters, e.g., network load. Distributed management, on the other hand, supports fine-grained adaptation, though with high demands on the physical support. Clustered-based management provides a tradeoff between the energy efficiency and the required physical platform.

From the chapter's study, we can identify that self-management operations on each architectural level have specific tradeoff between performance, energy and overheads, which points out that a systematic integration of these operations may exploit the benefits of all levels' adaptation. In particular, although distributed DVFS is shown to have better energy-performance tradeoff than clustered or centralized approaches, its limitations are still noticeable. In

Table 3.2: Summarizing Comparison of Centralized, Clustered and Per-Router DVFS in Energy-Performance Tradeoff

Architecture	Temporal Adaptation	Spatial Adaptation	Monitoring Communication Overhead	Complexity	Energy Efficiency
Centralized	Slow (in decision-making and switching	Not addressing locality	Congested monitoring network	Low	Low
Clustered	Moderate (due to the VR delay	Addressing intra-cluster traffic	No congestion very low energy overhead	Moderate (VR needed)	High
Distributed	Fast (due to power switches)	Very fine-grained (with possible oscillations)	No global monitoring communication	High (MPN required)	Highest (too frequent switchings incur penalties)

addition to oscillation, distributed decision-maker is not aware of global performance. For instance, the application or network latency can not be captured locally, as the latency depends on the status of multiple routers. The clustered architecture, on the other hand, has a per-cluster decision-maker that can be designed with cluster performance awareness beyond average traffic load (Chapter 5). A hybrid architecture, which combines the fast local reconfiguration and clustered decision-making, may exploit both of their benefits.

In the following chapters, a systematic paradigm to design self-aware and adaptive services on all architectural levels will be introduced. The paradigm, Hierarchical Agent-based Adaptation (H2A), will integrate the pros of different self-management architectures when providing run-time energy management and dependability.

Chapter 4

Hierarchical Agent-based Design Platform

Hierarchical Agent-based Adaptation (H2A) is a new design paradigm addressing the monitoring, decision-making and reconfiguration (MDR) processes. The previous chapters have motivated a dedicated design dimension for self-adaptation services in parallel embedded systems. This chapter starts to introduce H2A as a systematic paradigm to design SAA systems. The backbone of H2A is an agent hierarchy, from the top-level platform agent, the middle-level cluster agent, to the bottom-level cell agent. Each level of agents is responsible for services of different scopes, priorities and granularities. Such partition ensures that both high- and low-level services can be fulfilled without incurring bottlenecks. Each level of agents follows specific SW/HW co-design for implementation, so that the flexibility of services is provided without incurring non-scalable overhead. This chapter is an overview of the H2A paradigm, while the following chapters will present the architectural design and implementation.

4.1 Agent as a Design Abstraction

The term *Agent* originally comes from artificial intelligence and software engineering, with a diversity of meanings. One classic definition can be found in [97]: "an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators".

Based on the classic definition, we use *Agent* as a design abstraction, refer-

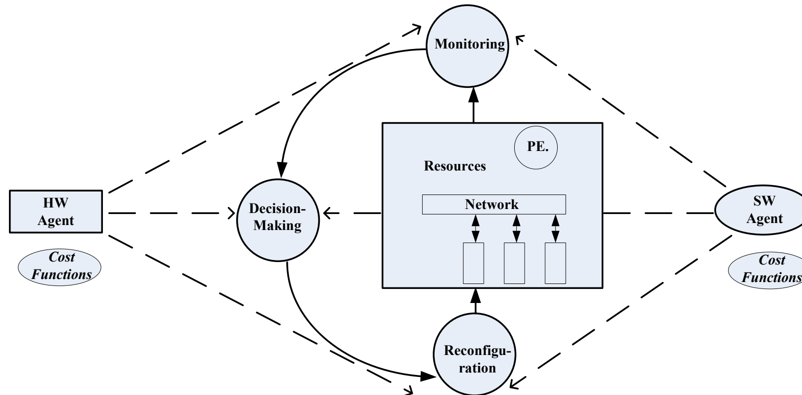


Figure 4.1: The Integration of Agents for Self-Adaptation on Processing and Communication Platforms

ring to any control entity in the MDR processes. Such an abstraction enables the separation of the design space into the processing, communication (conventional) and self-adaptation. In the self-adaptation subsystem, agents control all monitoring (upon the monitors) and reconfiguration (upon the actuators), and serve as the decision-makers (Fig. 4.1). With a separate self-adaptation design layer, the designers will consider the MDR processes as a unified subsystem generically applicable to different processing and communication platforms, instead of a set of ad-hoc techniques.

In terms of functions, agents share similar behavioral patterns. They observe (via monitors) and reconfigure (via actuators) resources. The monitoring operations provide self-awareness of the system. The agents process the information gathered from the monitoring operations with built-in cost functions (Section 2.1). Accordingly, they determine if and how reconfigurations should be performed on the resources, which lead to self-adaptation. If the system's environment is considered in the adaptation process, the agents also observe and act upon the environment. Between multiple agents, there is also communication to exchange information. In terms of implementation, agents can be realized as software, hardware, or any type of hybrid entities. There can be a single agent or multiple agents.

4.2 H2A Architecture

To integrate agents for monitoring, decision-making and reconfiguration operations in a scalable manner on parallel embedded systems, multi-layered agents with hierarchical scopes and priorities are constructed, i.e., Hierarchical Agent-based Adaptation (H2A)(Fig. 4.2). In this thesis, the agent hierarchy is composed of three levels: platform agent, cluster agent and cell agent.

To enable the self-adaptation of a system, firstly the application should be embedded with metadata, which includes the application progress and performance requirement. The information of the performance requirement sets the performance objective for the adaptation. Common requirements include timing constraints, e.g., soft/hard deadlines, and dependability requirements, e.g., MTTF (mean-time-to-failure). The application progress indicates the important timestamps for the platform agent to measure the performance. One example of specifying the application progress is *Application Heartbeats* [46], as illustrated in Fig. 4.3. In the program, certain locations are inserted with API functions (i.e., heartbeats), which indicate the progress of the application. By checking the timestamps of heartbeats, the system is aware of the execution speed of the application. The metadata can be specified by the application designers, and is independent of the platform that will run the application. The thesis does not provide in-depth exploration of the application design to integrate the meta-data, instead it assumes that such information is already embedded in the applications.

The platform agent is the top level monitor, which receives the meta-data from the application and accordingly configures the whole system at a coarse granularity. It is responsible for resource allocation, including mapping processors to application tasks, and network configuration. At run-time, it traces parameters indicating overall system performance, for instance, the total system power or the speedup. Based on these parameters, the platform agent may reconfigure coarse-grained system settings, for instance, reconfiguring the network topology. The coarse-grained monitoring operations handled by the platform agent usually require a large amount of data processing. For instance, power and performance aware application mapping, including dynamic mapping, goes through many iterations of searching of all resource information [20]. In addition to monitoring the platform, the platform agent is also monitoring the performance of the cluster agent. As agents themselves are also victims of

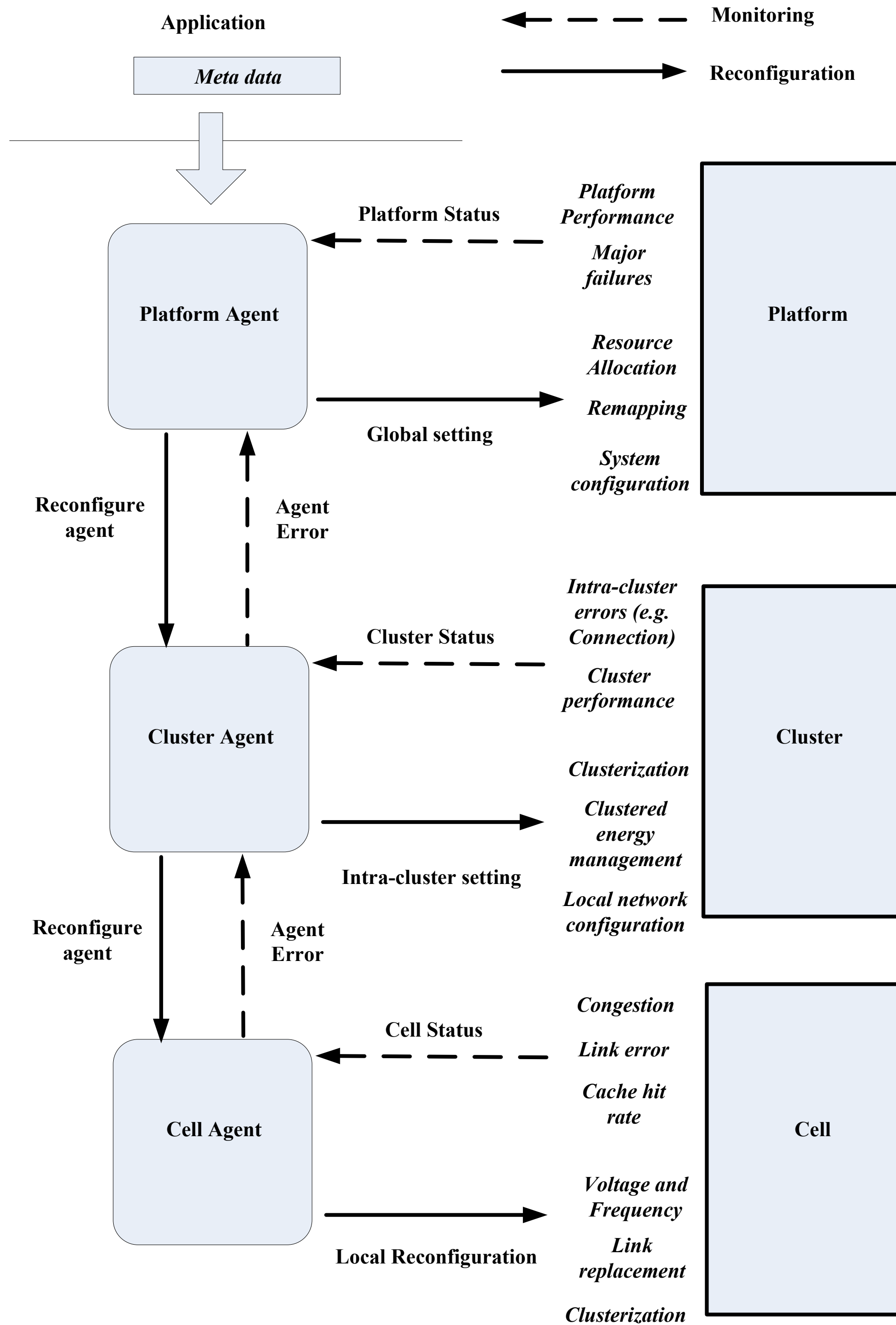


Figure 4.2: Multi-layered Agents for Hierarchical Monitoring, Decision-Making and Reconfiguration

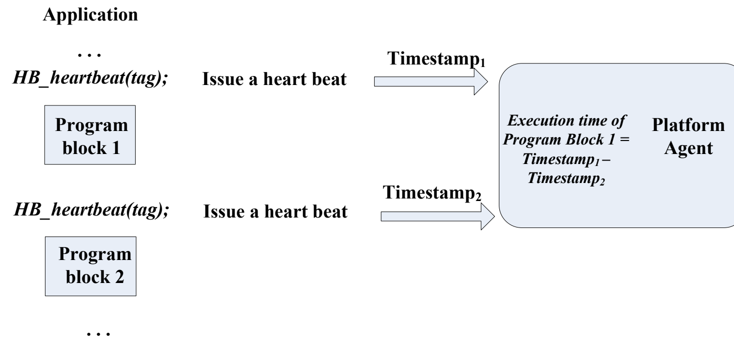


Figure 4.3: Application Awareness with Program Heartbeats

errors and failures, each level of agent is responsible for the fault-management of its lower-level agents.

Cluster agents work on the lower level of the platform agent. Each cluster agent is responsible for a region, i.e., a cluster, in the platform. The range of a cluster is design specific, for instance, a group of cores in the same voltage island on a many-core platform. In terms of monitoring, the cluster agent traces the performance of and the errors within a cluster, for instance, the network communication in one voltage island. In terms of reconfiguration, each cluster agent can only reconfigure the settings in the corresponding cluster. For instance, if each cluster has its own voltage and frequency controller, the cluster agent will be responsible for the controller's setting. In addition, all cluster agents need to follow the coarse-grained reconfiguration set by the platform agent. For instance, the platform agent can decide on a dynamic update of the clusterization (Chapter 6) with different cells grouped to the cluster. Then the cluster agent will be responsible for the newly formed cluster. In addition, the cluster agent is monitoring the cell agent, while being monitored by the platform agent.

The lowest-level agents, cell agents, work on a fine granularity. Each cell agent tightly monitors and reconfigures a cell, which can be a processor, router or a NoC node composed of one processor and one router. The partitioning of cells depends on the smallest granularity for adaptation in the system, and can be specified by the designers. Each cell agent monitors the parameters indicating the local status and errors. For instance, link errors can be detected by the cell agent with repetitive checking upon transient errors. The cell agent also reconfigures the local settings, if tunable parameters are given for each

cell. For instance, [115] presents a many-core platform, where each core can adjust its power scheme independently. In this case, such fine-grained power management lies in the functions of cell agents.

Following the discussion above, the functional partitioning of agents in H2A can be summarized as in Table 4.1. The functions of the platform agent, cluster agents and cell agents complement each other to perform the monitoring and reconfiguration operations on all architectural levels. Such partitioning provides performance (of adaptation services) scalability, as the low-level fine-grained operations are captured by the cell agents, thus alleviating the burdens on the platform agent. In terms of design productivity, the systematic partitioning avoids ad-hoc designs by providing a generically applicable monitoring and reconfiguration framework. The following chapters will present examples where various services can be designed with such a framework.

While Fig. 4.2 gives the basic form of agent monitoring and reconfiguration in H2A, additional features can be added as extensions. Firstly, in massively parallel systems, it is likely that one level of cluster still can not distribute the monitoring and reconfiguration roles efficiently. In this case, more levels of clusters can be added into the hierarchy in the form of *Superclusters*. In addition, the adaptation operations can involve multiple levels of agents, or flow across multiple levels (Table 4.1). For instance, a platform may have a certain amount of spare nodes. In case a cluster runs out of normal nodes, the cluster agent may ask for a spare node from the platform agent. If the platform agent decides to grant a certain spare node, it will configure the spare node to be included into the cluster. In this case, the platform agent directly configures a cell. In Chapter 6, related designs will be presented.

4.3 Agent Implementation Guidelines

Self-adaptation operations introduce implementation overheads. In order to ensure physical scalability of the proposed H2A architecture, SW/HW co-design is followed in choosing proper implementation alternatives for the self-adaptive operations. In particular, agents on different levels are implemented as software, hardware or hybrid, based on the diversity of functions, complexity of algorithms and timing requirements.

Table 4.1: Generic Functional Partition of Agents on the Platform

Level	Priority	Resource Monitoring and Reconfiguration	Monitored Agents	Features
Platform agent	Highest	Platform performance and energy; Major failures; Low-level status upon request	Cluster agents; Cell agents upon request	Coarse granularity; Global influence; Diverse functions; Long-latency operations
Cluster agent	Medium	Cluster status; Energy-performance tradeoff; Local network configuration; Low-level performance upon request	Cell agents	Intra-cluster influences Moderate amount of processing
Cell agent	Lowest	Cell reconfiguration; Local errors	None	Fine granularity Low-latency, fast response Minimal amount of processing

4.3.1 Alternatives

Agent functions can be implemented with various software and hardware alternatives, including dedicated processors, time-sharing on processors, reconfigurable micro-controller, or custom hardware.

When implemented as an embedded general-purpose processor, an agent can perform diverse monitoring and reconfiguration operations as software functions. For example, ElastIC architecture [113] adopts a single processing unit for dynamic testing operations and as the global-level scheduler. A software agent has high flexibility as the instructions can be reloaded at run-time to reconfigure the functions. In case of complicated algorithms, such manner of agent design saves the area overhead compared to purely hardware-based design, but it is usually slower than dedicated hardware implementation.

As another alternative, sharing a physical processor with the data processing saves the physical overhead compared to a dedicated processor for the agent. In Chapter 7, an example of a cluster agent running as a thread on a general purpose processor will be presented. However, such an alternative introduces design complexity, for instance, the coupling of the errors of the data processing and the agent function. If the processor fails, both data processing and the agent function may stop working properly.

Micro-controller is another alternative for agent implementation [17]. It has a much simplified processing core compared to a general-purpose processor, thus is low in complexity. It can run different instructions (though with limited memory), thus provides more flexible functions than custom hardware. Micro-controller is a suitable trade-off between flexibility and overheads for agents with low-complexity monitoring algorithms. It is also faster than a general purpose processor.

The fourth alternative is custom hardware. Implementing agents as dedicated hardware supports low-latency operations and minimizes the area overhead compared to the same number of software-based agents (e.g., a processor). It is suitable for agents with very simple and standardized operations. When the parameters for monitoring and reconfiguration are similar for different applications (e.g., link error, network load, latency counter), the hardware-based agent can be wrapped into an IP block with the processing or communication component.

Table 4.2: Agent Implementation Guidelines

Level	Complexity	Timing	Implementation
Platform Agent	Highly complex; Various system configuration algorithms	Long processing time; Latency-tolerant	General-purpose processor (possibly with additional hardware logic)
Cluster Agent	Moderately complex; Simpler algorithms; Less diversity	Shorter than platform-level algorithms	SW/HW co-design
Cell Agent	Simple; Dedicated operations	Low-latency; Frequently issued	Hardware; Micro-controller

4.3.2 SW/HW Co-Design for Agents

Based on the pros and cons of each alternative, the implementation guidelines for each level of agents can be discussed. The major considerations are the diversity of monitoring functions, the operations' timing requirements and complexity of algorithms (Table 4.2).

Platform agent, at the top level, has a wide range of monitoring operations and needs to perform complex processing for global optimization. A software implementation with general-purpose processor provides the flexibility and reconfigurability. In addition, the global-level reconfiguration is usually allowed very long latency, for instance, the mapping process. Thus the speed disadvantage of general-purpose processor is not an issue. Such manner of implementation is also scalable in terms of physical overhead as only one such processor is required for any-sized platforms. The software agent may require certain hardware circuits for pre-processing (Chapter 6 will present an example on this).

The cell agent, distributed for each fine-grained functional component, is suitably implemented in custom hardware or micro-controller. Firstly, low-level services, for instance, fault detection and network load collection, require fast operations. Custom hardware or micro-controller-based agents provide the required fast operation. In addition, low-level services are usually simple and easily modularized. For instance, a set of sensors (e.g., load, temperature, fault) controlled by one cell agent can be added to a network node. Generally speaking, hardware-based implementation fulfills the simple operations

in the low level with much lower overhead than a general-purpose processor. Since each cell requires a cell agent, a software-based approach is infeasible for parallel embedded system.

The complexity of cluster agents lie in between the platform and cell agents, thus their implementation is design-specific.When the cluster agent is responsible for simple monitoring operations, it can utilize hardware-based approach. Chapter 5 will present a case study where the cluster agent is performing DVFS with hardware logic. When the cluster agent handles a diversity of monitoring operations, a software implementation is more suitable. Currently, some software-based approach is already proposed even for each individual core. For instance, [48] allows operating system threads to be loaded for each core. With the constant increase of cores on a single chip, the size of current platform may be comparable to a cluster in the future platform, thus a software-based approach for cluster-level agent is reasonable for large clusters.

4.4 Agents and Operating System

A computer-based system is generally composed, from the highest-level, of application, operating system[1] and hardware. Based on this generic system partition, the software agents belong to the function of operating system, while the hardware agents and the affiliated components provide the architectural support to the operating system. While the thesis does not aim to design an operating system, a discussion on the positioning of the agent architecture in the general computer system is necessary.

As illustrated in Fig. 4.4, the hierarchical agent architecture can be integrated into a multiprocessor operating system with hierarchical prioritized processes for monitoring and reconfiguration services. The design needs to address the following considerations:

- Software agents which manage the coarse-grained adaptation services (e.g., resource allocation, global network configuration) should be designed as multiple processes. These processes need to have hierarchical priorities, as the platform agent and cluster agents have different scopes in the system management.

[1] Discussion on the relation/difference between operating system and middleware is omitted, as such differentiation is beyond the scope of this thesis.

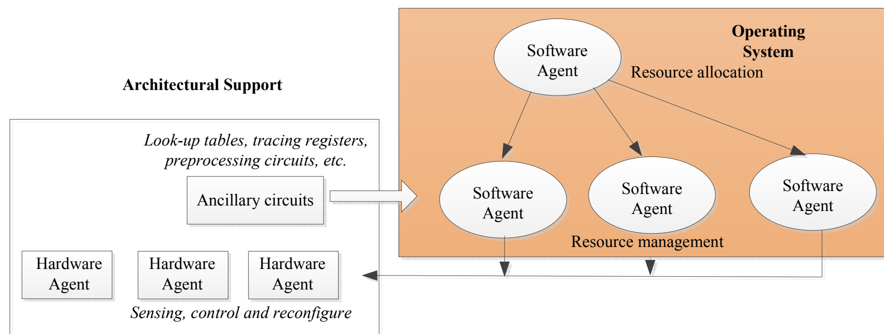


Figure 4.4: Agents and Operating System

- Hardware agents, in particular the lowest-level cell agents, realize uniform and simplified functions, for instance, load monitoring, voltage switching, etc. With less flexiblity than software-based agents, they enable fast processing of predetermined monitoring and reconfiguration functions. Microcontroller or reprogrammable technologies improve the flexiblity of hardware agents by allowing limited manners of reconfiguration.

- Hardware supports for software agents are needed as ancillary circuits to enable the adaptation services. For instance, status look-up tables will be designed in Section 6.2 to support the dynamic clusterization and run-time resource management of platform and cluster agents.

- Regardless of software or hardware agents, they are both orthogonal in function to data computation and communication services (Fig. 2.4), as they are monitoring and reconfiguring the normal application operations.

As a proof-of-concept prototype, Chapter 7 will implement a software-based platform agent, thread-based cluster agents with specific hardware support, and hardware-based cell agents for modularized fine-grained services. Though the implementation is based on distributed shared memory architecture, the paradigm itself is not confined to any parallel/distributed operating system architectures.

4.5 Related System Architectures

Multi-agent-based or SW/HW co-design control systems for monitoring and self-adaptation services have been proposed in previous works [82, 83, 113, 24, 2, 21, 109, 50]. A representative set of these related system architectures are discussed below and compared with H2A.

Some works address generic system architectures for run-time adaptation. For instance, [82, 83] propose multi-agent control system on reconfigurable platforms. They suggested that both software and hardware agents can be modeled with their beliefs, desires and intentions. The agents implemented on reconfigurable hardware were demonstrated to be much faster than a software implementation, for a sensor fusion system. In H2A, software and hardware agents co-exist and are chosen based on the complexity and flexibility of the functions, the speed requirement and the silicon overhead. The agent hierarchy not only relates to the functional scope of each level of agents, but also their implementation style (Section 4.3). Another work, ElastIC [113], is an adaptive system architecture for self-healing in many-core System-on-Chip. A centralized DAP (diagnostic and adaptivity processing) unit is supervising a pool of adaptive processing elements. Each processing element contains observable and tunable parameters, for instance, power monitors. The DAP unit dynamically tests (e.g., by taking each processing element offline) and reconfigures cores with degraded performance. However, this work does not provide in-depth architectural exploration for different adaptive services or the implementation of the system architecture. In [109], a system architecture is presented for evolvable systems, partitioned into application, operation system and hardware architectural layers. This work compares using software or hardware implementation libraries for self-adaptive operations (observe, decide and act). In comparison, H2A emphasizes on utilizing both software and hardware agents upon parallel embedded systems, while the libraries proposed in [109] can be integrated in H2A for realizing agents.

More efforts on self-adaptive architectures have been proposed to address specific components, functions or services. A two-level controlling architecture is presented in [24]. In this architecture, a software-based global manager determines the management policies of the whole network. Each hardware-based delegate manager decides on the local reconfiguration based on the management policies. The architecture is mostly applied to building self-adaptive

network interface, without the discussion on supporting generic monitoring and reconfiguration services. In [2], a two-level agent architecture is proposed for run-time application mapping. The lower-level agent, cluster agent, is responsible for mapping within each cluster. The higher-level global agent stores resource utilization information for selecting and organizing clusters. The work focuses on a specific type of run-time operation, application mapping, without exploring other services. In addition, the work mainly addresses the algorithm design of the agent functions, without exploring the SW/HW implementation of the agents. [21] proposes a monitoring-aware system architecture and design flow for NoC. It presents hardware-based probes for transaction debugging and QoS (Quality-of-Service) provision. H2A, in contrast, emphasizes on a widely-applicable hierarchical control backbone for both functional and non-functional design goals, e.g., energy efficiency and dependability.

Compared to existing generic system architectures (e.g., [82, 83, 113, 109]), H2A presents in-depth architectural exploration for various adaptive services, including energy management (Chapter 5) and dependability (Chapter 6). In addition, different levels of agents are implemented in a self-aware and adaptive Network-on-Chip (Chapter 7) with quantitative discussion on the physical scalability. Compared to works addressing specific self-adaptive services or algorithms (e.g., [24, 2, 21, 50]), the focus of H2A is to propose a generic and compatible system architecture for various coarse- and fine-grained services. Hence a hierarchical agent-based control backbone is integrated. In addition, agents can be either software, hardware, or SW/HW co-design, instead of being limited to one specific type of implementation.

4.6 Chapter Summary

This chapter presents H2A as a generic design paradigm and system architecture for SAA systems. It firstly introduces agent as a design abstraction to separate the design concerns for adaptivity from conventional processing and communication infrastructure. Then it presents the agent hierarchy, the functional partitioning and the implementation guidelines of agents. The functions of each agent level is partitioned based on the scope of the specific level, in order to distribute the monitoring and reconfiguration workload. The implementation of agents follows the SW/HW co-design approach to provide physical scalability while supporting the function of each agent level.

With the generic H2A architecture explained, from the next chapter, the thesis elaborates on the architectural design and implementation of agents for various monitoring and reconfiguration services. The implementation of agent functions as software, hardware or SW/HW co-design will be demonstrated respectively with discussion on their performance, efficiency and overheads.

Chapter 5

Coarse- and Fine-Grained Energy Management

Energy management is one of the major services in self-adaptive systems [39], due to the stringent power and energy budget in parallel embedded systems. Energy saving or energy-performance trade-off can be provided on different system levels, from coarse-grained resource allocation to low-level buffer or link reconfiguration. Most existing works address monitoring or reconfiguration techniques on specific levels. A hierarchical approach, where energy efficiency is contributed by all architectural levels, is presented in this chapter, following the H2A design paradigm. The platform agent is designed as a software module, performing energy-aware mapping of applications onto the system. Within each cluster, dynamic energy-performance trade-off is performed by the cluster agent via directly monitoring the execution time of the application(s). The direct performance monitoring is enabled by the cell agents, which trace application milestones and report them to the corresponding cluster agents. In addition, each cell agent monitors and reports the local network load to the cluster agent. Architectural design and simulation will be presented to evaluate the effectiveness and efficiency of the proposed hierarchical agent-based energy management architecture.

5.1 System Architecture

To improve the energy efficiency, or dynamically fine-tune energy-performance trade-off, diverse system, architectural and circuit-level approaches can be uti-

Table 5.1: Monitoring and Reconfiguration Techniques for Energy-Efficient NoCs

Recon-figuration	**Monitored Parameters**	**Power/Energy Saving**
Dynamic voltage and & frequency scaling (DVFS) [115, 37, 14]	Time slack; Link/buffer utilization; Traffic congestion	Dynamic power/energy; Static power/energy
Dynamic buffer allocation [84]	Traffic congestion	Dynamic power/energy Static power/energy
Energy-aware mapping [49]	Communication volume; Hop counts	Dynamic energy
Clock gating	Activity prediction	Clock power
Power gating [110]	Activity prediction [80]/ Activity tensity [110]	Static power/energy
Driver and Receiver reconfiguration[86]	Current	Link power/energy

lized. For instance, on many-core systems, methods addressing the power/energy consumption of different components (such as processing elements, routers, memory) have been proposed (Table 5.1), from high-level resource allocation (e.g., energy-aware mapping) to low-level circuit techniques (e.g., driver reconfiguration). As can be observed from Table 5.1, most existing approaches address the energy saving on specific architectural levels by monitoring particular activity or status parameters. The H2A paradigm enables the design of a hierarchical energy management backbone, which exploits the energy saving of multiple architectural levels. Fig. 5.1 overviews such a system.

As illustrated in Fig. 5.1, the system allows for distributed power reconfiguration. Multiple power lines of different supply voltages are laid out on the chip (Section 3.2.1). Each processor or router can be switched to one of the power supplies via power switches, which enables fine-grained voltage reconfiguration. In addition, the system supports distributed frequency switching, with a frequency synthesizer attached to each network node. Given such GALS (globally asynchronous locally synchronous) timing, a bi-synchronous FIFO [81] is inserted on every inter-router channel.

Upon the power reconfigurable many-core system, the partitioning of cells and clusters can be performed as follows. Each cell includes a router and

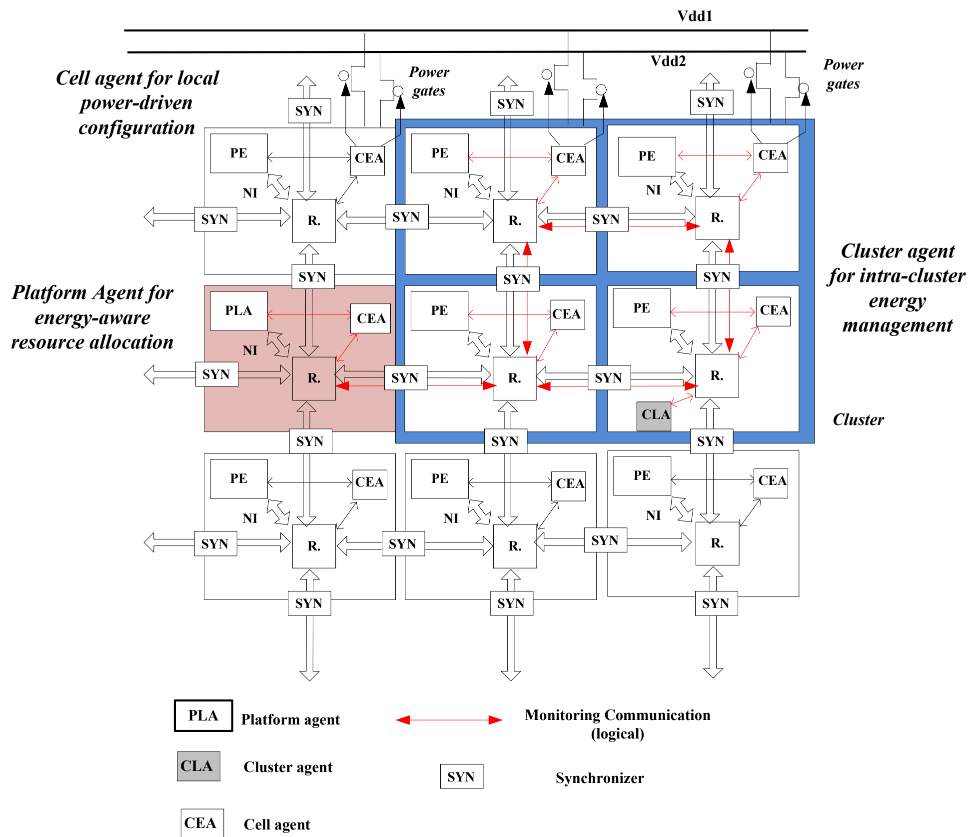


Figure 5.1: Hierarchical Agents for Energy Management upon Distributed Reconfigurable Platform

its connecting links, the corresponding processor, and the memory. With distributed power switches and frequency synthesizers, the voltage and frequency of each cell can be individually configured without influencing the operation of other cells. Every cell is monitored by a cell agent, which is a hardware module attached to the router. Each cluster is assigned as a group of cells. The cluster agent can be a software- or hardware-based module running in one cell. The cluster agent communicates with any cell agent within the cluster via the monitoring network (Section 3.1.4). The platform agent, hosted by one processor, runs software instructions. The platform agent can configure any cell or cluster with the monitoring network. The partitioning and SW/HW design of agents in Fig. 5.1 imply no restriction from the design paradigm's perspective, but serve as a demonstrating example (for instance, in Chapter 7, a hybrid cluster agent with both software and hardware parts will be designed and implemented).

Three-level operations are designed for each level of agents, to achieve hierarchical energy management. As the coarse-grained controller, the platform agent performs energy-aware mapping, which has system-wide influence on the energy consumption. Each cluster agent performs CDDR (Clustered Decision-Making, Distributed Reconfiguration), to adaptively reduce the energy consumption in the local cluster by directly monitoring the application performance. CDDR is supported by the hybrid monitoring service of each cell agent. In particular, both the local router traffic load and the application milestones are traced by each cell agent, and reported to the cluster agent. Superior to the clustered and distributed architectures in Chapter 3, the hybrid CDDR technique combines the energy saving potential of distributed reconfiguration and the performance awareness of the cluster agent. The choices of energy-saving reconfiguration- mapping and DVFS, are exemplified techniques among the vast design space of energy-efficiency communication (Table 5.1). The system architecture itself is not limited to these reconfiguration methods. The quantitative study will focus on energy saving of on-chip networks (routers and interconnects). Both energy-aware mapping and DVFS significantly influence the communication energy [49, 37]. The following sections elaborate the design of each agent and its energy management operations.

5.2 Energy-Aware Application Mapping

Application mapping is a coarse-grained resource allocation technique. Energy-aware mapping allocates application tasks to the processing elements, in a way that minimizes the communication energy between the processors [49]. It is effective in energy reduction, as [49] reports 51.7% saving in energy consumption with a branch-and-bound algorithm compared to random mapping.

In a NoC system where applications are expected to be loaded at the run-time, we are considering a scenario where multiple applications are to be mapped onto different areas dynamically. With an increasing number of resources integrated on a chip, multiple applications can simultaneously run on a single chip. For instance, on a 167-core computing platform [115], 9 cores are used to realize a JPEG encoder, and 15 cores are utilized to implement a H.264 encoder. Such mapping is performed by the platform agent, which needs to know the application graph (or the application characterization graph in [49]) and the available resources before mapping. The application graph, in particular the inter-task communication graph, is a set of triples $< T_i, T_j, V_{i-j} >$. Each triple denotes the communication volume V_{i-j} from task T_i to T_j. The application graph can be encapsulated in the application as metadata, and stored in the system memory, which is accessible by the platform agent. The resource availability information can be logically stored as a SLT (status look-up table), where each entry stores the run-time status of a node.

The system architecture has no limitation on the applicable mapping algorithms, which are design-specific. Since application mapping is in general NP-hard [49], fast algorithms are desirable for run-time mapping where the timing is a major consideration. Here we exemplify an in-house two-step algorithm for the energy-aware mapping process [127], illustrated by Fig. 5.2. The first step, MER(maximal empty rectangle)-based application mapping, finds suitable rectangular areas for each application. The second step, tree-model-based task mapping, assigns processors to tasks in each application. As modeled by Eq. 5.1 ([49]; E_b is the communication energy of one bit; E_S and E_L are the communication energy of the bit traversing the switches and links respectively), the dynamic energy consumption is proportional to the hop counts of packets. Hence both steps aim to reduce the hop counts of inter-processor communication.

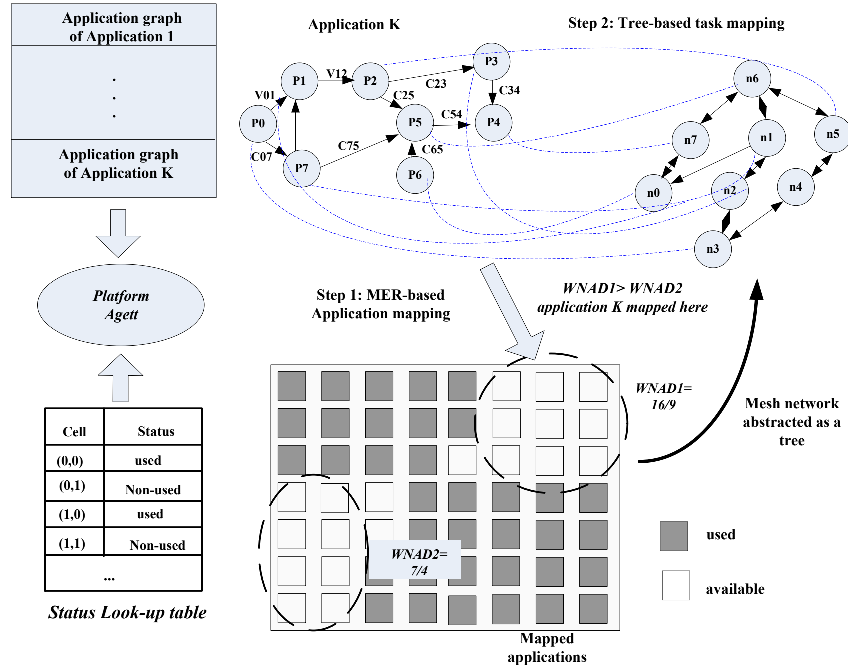


Figure 5.2: Exemplified Two-Step Energy-Aware Mapping Algorithm on Platform Agent

$$E_b = n_{hops} \times E_S + (n_{hops} - 1) \times E_L \tag{5.1}$$

In the first step, MER is the largest available area in the shape of a rectangle. Choosing MER is an effective technique originating from the FPGA mapping problem [7] to significantly reduce the exhaustive search space. To choose a proper rectangle, we firstly consider a parameter NAD (node average distance; $NAD = \frac{X+Y}{3} \times (1 - \frac{1}{X \times Y})$), which represents how close on average the processors are in a specific rectangle. X and Y are the number of nodes on the two dimensions of the rectangle. The smaller NAD is, the more likely the communication energy after mapping is low. The parameter NAD does not take into account the situations where no single rectangular area has enough nodes for the application, thus a modified parameter $WNAD$ (weighted node average distance; $WNAD = max(\frac{N_{tasks}}{N_{nodes}}, 1) \times NAD$) is considered. The weight $\frac{N_{tasks}}{N_{nodes}}$ represents how completely an application can be mapped on a rectangle. A contiguous area is always favored compared to several split-up areas, considering the additional energy needed for communication in-between areas. Therefore, the application-mapping step searches for the available rectangular

area(s) with the highest $WNAD$, which is a fast (but not necessarily optimal) mapping solution.

In the second step, tree-model-based task mapping starts with the transformation of a mesh network into an extended tree (Fig. 5.2). In a nutshell, the transformation starts with choosing the center point of the network as the root node of the tree, which has the shortest average distance to other points in the network. The neighbors of the center point are put as the children nodes of the root node (from left to right). The procedure continues until all points in the network are put onto the tree. Then tasks in the application are mapped onto the extended tree. We first place the process with the highest communication volume onto the root node, in order to reduce the average communication distance. Then we place the process that has the highest amount of communication with the already-mapped processes, onto the highest available node on the tree (from left to right). The procedure iterates until all processes are mapped. The details of the application mapping and task mapping algorithms can be found in our previous work [127]. Tree-model based mapping, due to its simplicity, is significantly faster than exhaustive search or other existing algorithms, and achieves satisfying results in reducing energy consumption [128].

5.3 Intra-Cluster Energy Management

While the platform agent allocates the application tasks onto the processors, run-time energy management of each application is performed by the cluster agent, based on the actual traffic and workload. In Chapter 3, the decision-maker in each cluster performs cluster-wide DVFS, which adapts to the average traffic load in the cluster. Cluster-wide reconfiguration can not adapt to the traffic variations in different locations of a cluster. In addition, a voltage-regulator-based voltage switching incurs significant delay (Chapter 3). To address these issues, CDDR (clustered decision-making, distributed reconfiguration) provides fine-grained power-driven reconfiguration (on the same scale as distributed reconfiguration), while directly monitoring the cluster performance.

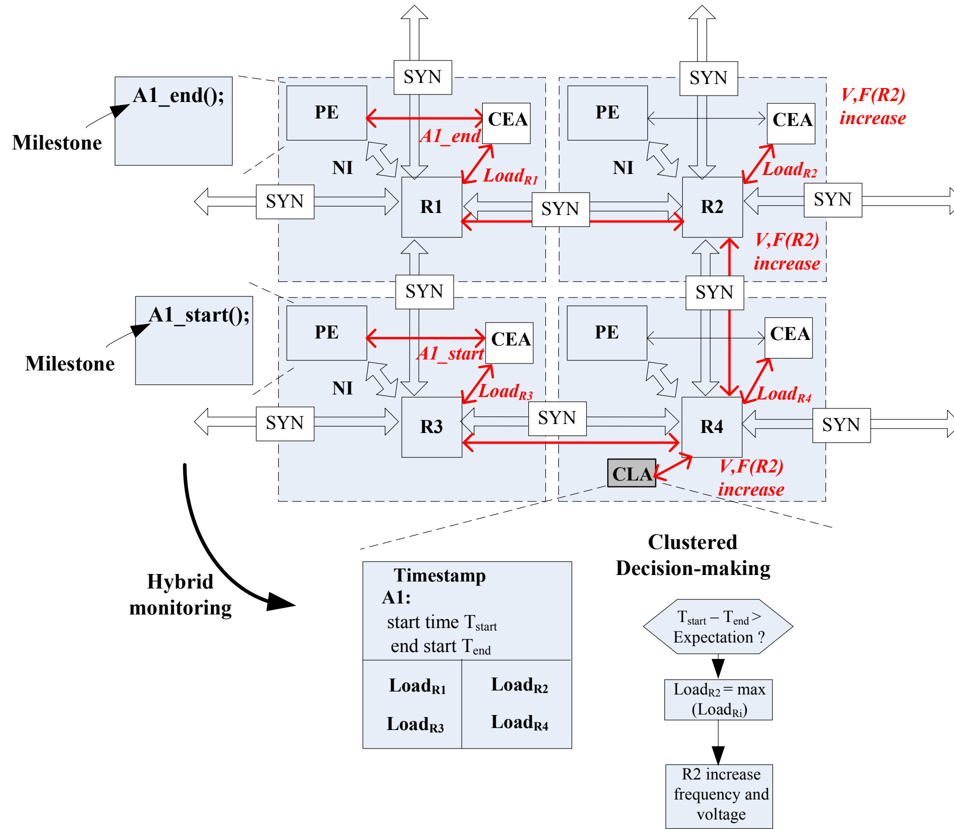


Figure 5.3: Overview of CDDR Technique for Intra-Cluster Energy Management

5.3.1 Cluster Architecture Overview

Fig. 5.3 illustrates the intra-cluster energy management technique, CDDR, with two levels of agents- cluster and cell agents. Each cell agent is tracing the traffic load and the occurrences of monitored events (milestones), and report them to the cluster agent. The cluster agent traces the timestamps of the application milestones, and accordingly evaluates the application performance in the cluster. Based on the local traffic load, congested network areas can be identified. Then the cluster agent attempts to reconfigure the cluster, in particular the congested areas, so that the energy consumption is minimized while the application performance is within the requirements. Briefly, CDDR relies on hybrid monitoring of local parameters (e.g., load) and cumulative parameters (e.g., latency), and performs distributed reconfiguration with a clustered decision-maker.

5.3.2 Load and Latency Monitoring

The cluster and cell agents provide load and latency monitoring in the cluster. Local network load (e.g., the buffer occupancy) is traced by the cell agent, and reported to the cluster agent. The occurrences of milestone instructions (such as the first or last instruction of an application or an application stream) are also traced by the cell agent, and sent to the cluster agent.

Such hybrid monitoring is designed to compensate for weaknesses in conventional load monitoring. The performance of network communication is often indirectly measured by load-related parameters, including the occupancy of buffers [90, 71] or the utilization ratio of links [107]. Generally, the higher load the network has (after the initial warm-up period), the longer average latency the packets have due to more conflicts in the transmission. However, the relation between these parameters and the actual performance varies with different network topologies, run-time traffic patterns, dynamic flow control schemes and unpredictable fault conditions. Fig. 5.4 illustrates an experiment of three traffic traces: uniform, b-model and hotspot (Section 3.3.2), where the relation between the latency and buffer load is shown. It can be observed that the same buffer load can result in drastically different latencies. Several causes can contribute to this variation. Firstly, the traffic pattern influences the relation between the buffer load and the latency, as shown in the simulation of the three traffic traces. In addition, when a high buffer load persists when the network is congested, the latencies start to steadily increase, as can be observed in the case of b-model traffic. From the analysis, it can be concluded that the load monitoring is not a reliable technique if the system is dynamically reconfigured.

The load monitoring is realized with counters in each router, performed by the corresponding cell agent (Fig. 5.5). For each buffer queue (input from the processing element and the north, south, west and east neighbors) in the router, the difference between the head and tail pointers is the number of buffers occupied. The total number of occupied buffers in each router indicates the traffic load of the cell. The average of such traffic load in one period of time (the monitoring window) is sent to the cluster agent via the monitoring network.

The latency monitoring is enabled by tracing application milestones, which are instructions denoting the progress of the application, e.g., the first and the last instructions of an application indicating its starting and ending times. As

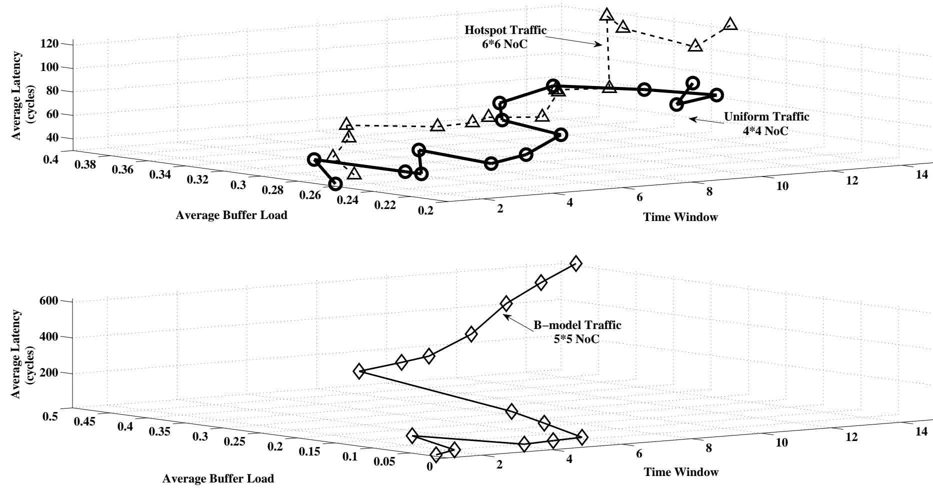


Figure 5.4: Relation between Buffer Load and Latency (in Cycles) of Three Traffic Traces in Different Time Windows

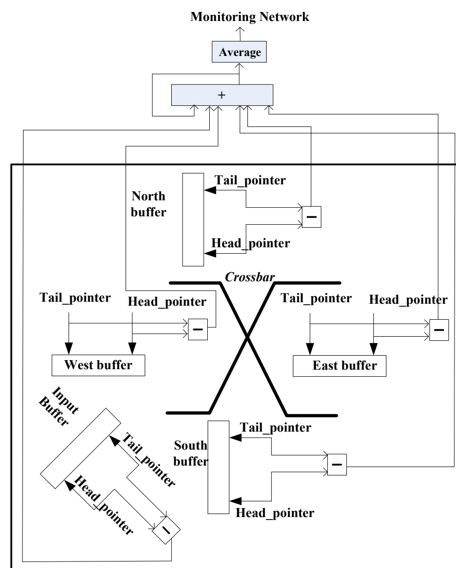


Figure 5.5: Monitoring Load in Each Router

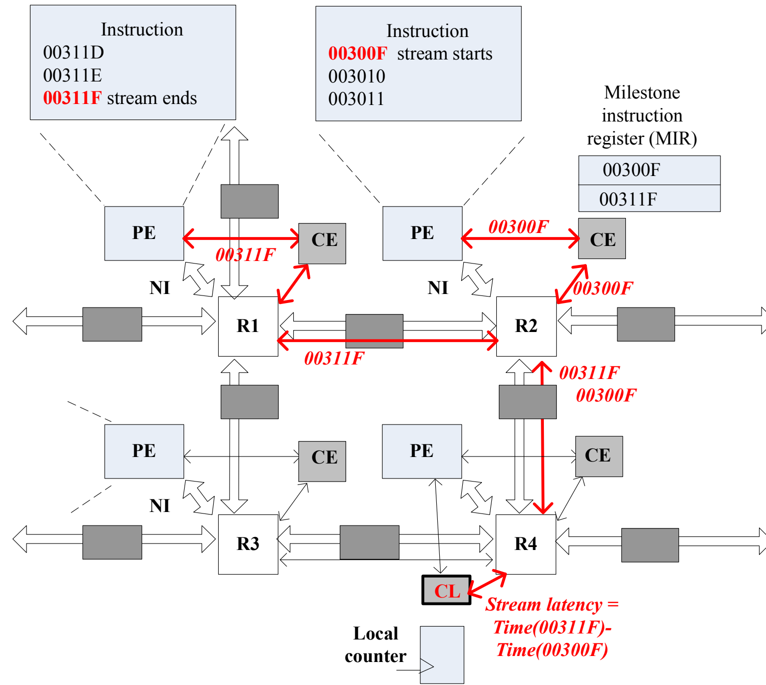


Figure 5.6: Latency Monitoring with Cluster and Cell Agents (exemplifying one MIR for brevity

illustrated by Fig. 5.6, the cell agents will be informed by the upper level agents (the cluster or platform agent) of the instructions to be traced. The milestone instructions are stored in a rewritable MIR (Milestone Instruction Register). When such an instruction is executed, the cell agent sends a message to the cluster agent along with the instruction address. When the message is received, its timestamp is noted down by the cluster agent via checking the local clock (of the cluster agent). By subtracting the starting time of an application from its ending time, its latency can be obtained. As the monitoring network is designed with low traffic volume (no congestion), the transmission times of the starting and ending messages are similar thus are canceled out in the subtraction.

5.3.3 Clustered Decision-Making, Distributed Reconfiguration

With both latency and load monitoring, the cluster agent performs CDDR within each cluster. In contrast to the clustered architecture in Section 3.2.3, CDDR is able to individually reconfigure each node. In contrast to the dis-

tributed architecture, CDDR has a cluster agent that decides on the reconfiguration of all nodes within the cluster.

With the example of energy management, the flow graph of CDDR is illustrated in Fig. 5.7. The system is initially configured with the highest voltage and frequency available, to ensure all performance requirements are met. When a new application stream starts running, the cell agent traces the milestone instructions and the local load. With the timestamps of milestone instructions, the cluster agent firstly evaluates if the performance actually deviates from the requirements ($L_r > L_u$ or $L_r < L_l$; L_r,L_u and L_l are the run-time actual latency, upper boundary of latency and lower boundary respectively). If performance change is needed, the cluster agent decides on the cells to be reconfigured. For instance, if the latency is higher than the upper boundary, the cluster agent may increase the voltage and frequency of the most loaded routers, in order to reduce the latency. After the corresponding cell agents have reconfigured the routers based on the cluster agent's decision, the next application stream starts running.

5.4 System Integration and Quantitative Evaluation

To quantitatively evaluate the energy management with hierarchical agents, a set of self-adaptation operations to realize platform-level application mapping and cluster-level CDDR, are experimented with the NoC simulator (Appendix A), as summarized in Table 5.2. To analyze the influence of platform-level reconfiguration on the energy consumption, energy-aware mapping (Section 5.2) is compared with random mapping. To analyze the influence of cluster-level reconfiguration, CDDR is simulated and compared with fully distributed DVFS (Section 3.2.4).

5.4.1 Experimental Setting

A 6×6 NoC is simulated, divided into four clusters (each being a 3×3 mesh). The network is configured with the same setting as in Section 3.3.1. Briefly, the network has physically separate data channels (32-bit) and monitoring channels (8-bit). Each router can be dynamically reconfigured with voltages and frequencies, thus FIFO-based synchronizers are inserted on every channel. Two pairs of voltage and frequency values, (V_H=2V,F_H=1GHz) and(V_L=1.05V,F_H=$\frac{1}{3}$GHz) [14], are provided. The voltage switching delay

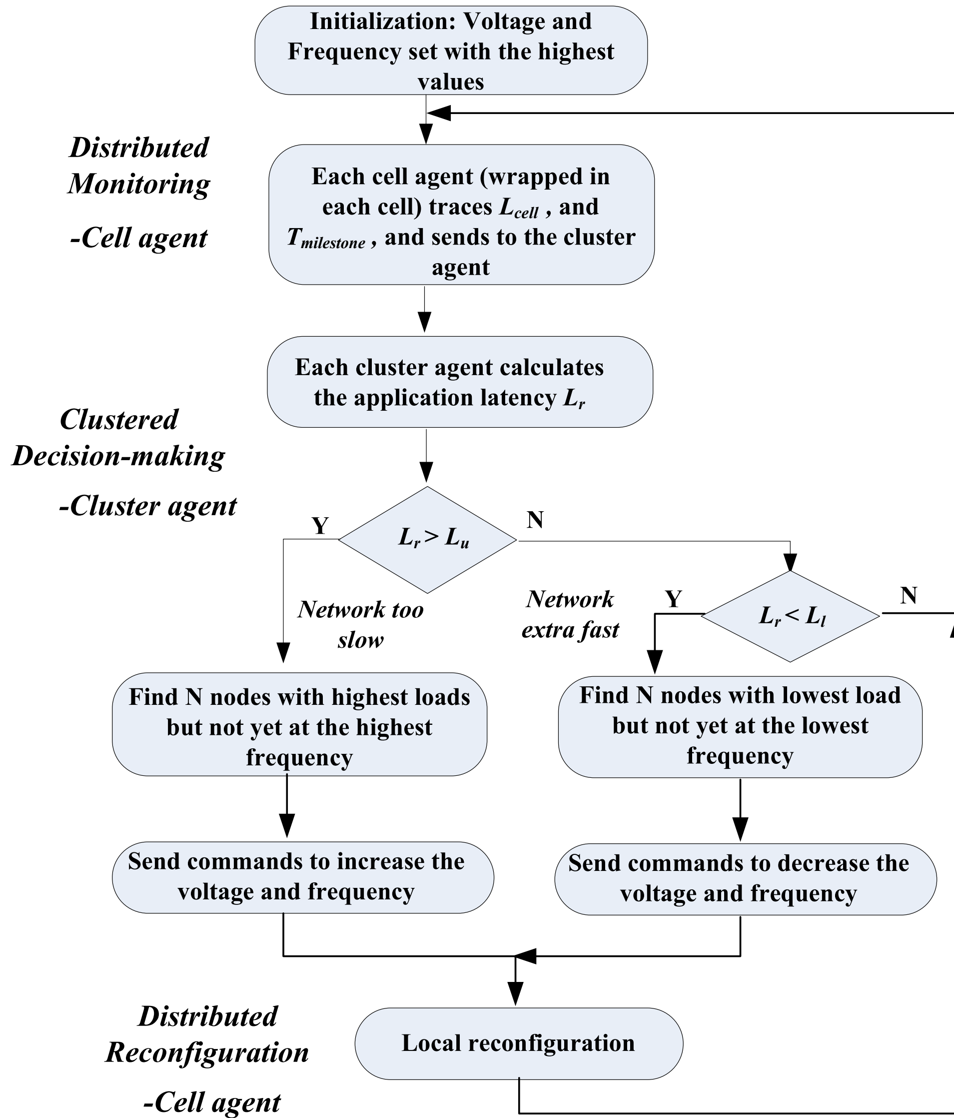


Figure 5.7: Flow Diagram of Clustered Decision-Making with Hybrid Monitoring

Table 5.2: Agents' Operations for Coarse- and Fine-Grained Energy Management

Agent	**Operation**	**Target**
Platform agent	Application mapping; Task mapping	Dynamic energy reduction for multiple applications
Cluster agent	Monitoring the latency of each application; Monitoring the traffic of each router; Decision-making for distributed DVFS	Cluster performance awareness; Local status awareness ; Intra-cluster energy-performance trade-off
Cell agent	Trace and report the router load; Trace and report milestone instructions; Actuate voltage and frequency reconfiguration	Local status awareness; Facilitate cluster performance awareness; Enable reconfiguration

for each router is 10*ns* (10 cycles at the higher frequency; details in Section 3.3.1).

Communication graphs of two applications from E3S benchmarks [28] are used for simulation. The first application (AUTO) is an auto industry benchmark, requiring 9 processors. The second application (CON) is a consumer electronic system benchmark, requiring 7 processors. Each of the four clusters in the NoC respectively runs the AUTO application with energy-aware mapping, AUTO with random mapping, CON application with energy-aware mapping and CON with random mapping (Fig. 5.8). For the CON application, two nodes will be left from mapping, which can be used as spares for fault-tolerance (to be studied in Chapter 6).

5.4.2 Results

For each of the clusters, CDDR is simulated and compared with distributed DVFS (DDVFS) and constant high voltage/frequency setting (CH). DDVFS follows the same architecture as in Section 3.2.4. Briefly, each cell agent is monitoring the local router's load, and changing the voltage/frequency based on the upper and lower thresholds. If the load (average in a monitoring window) is higher than the upper threshold, the router is assigned with the high

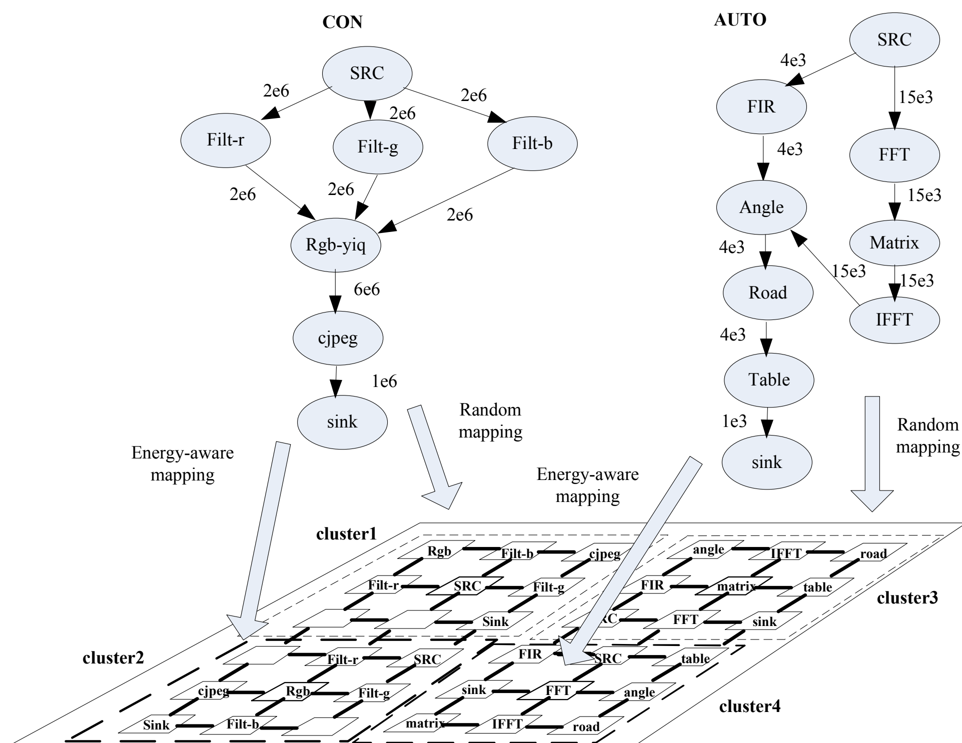
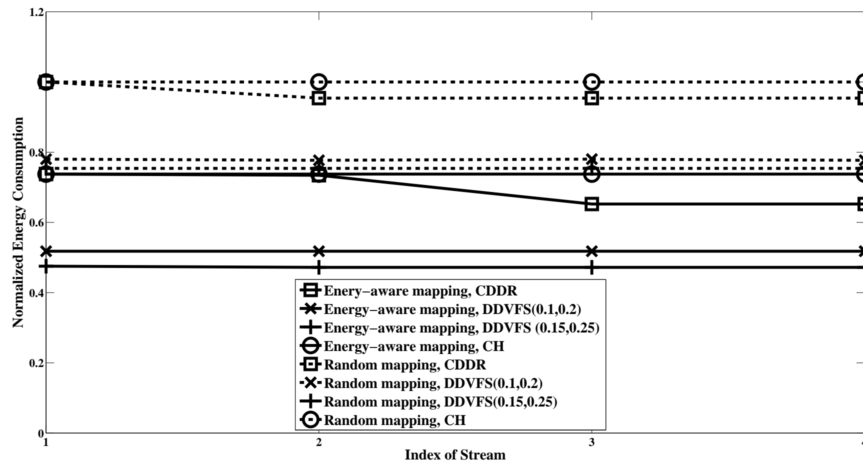


Figure 5.8: Mapping Two Applications on Four Clusters in a 6 × 6 NoC

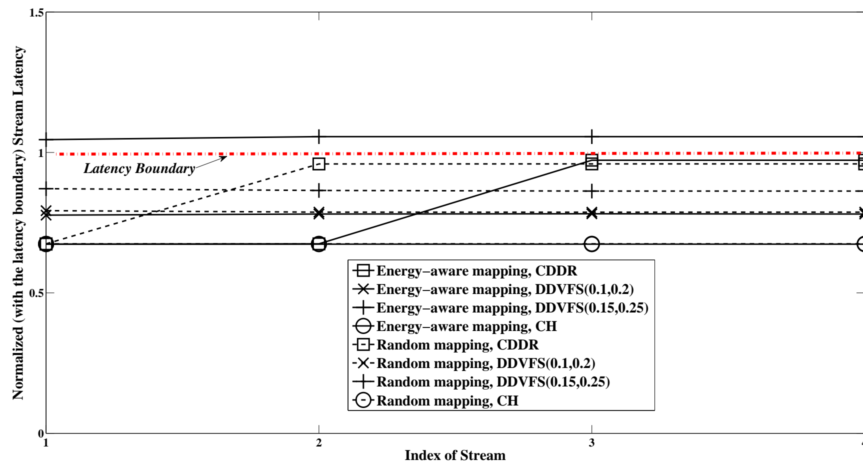
pair of voltage and frequency. If the load is lower than the lower threshold, the low pair of voltage and frequency will be applied. Two sets of upper and lower thresholds are experimented in the simulation: (0.1,0.2) as used in Section 3.3.3 and (0.15,0.25) to analyze the influence of load thresholds on DDVFS. CDDR follows the algorithm in Fig. 5.7. The upper boundary is the actual latency boundary expected for one application stream. The lower latency boundary is set as 90% of the upper boundary, so that the system can be reconfigured before the actual latency constraint is reached. In addition, leaving a margin between two boundaries reduces the chances of oscillation (i.e., the latency varies between a set of figures without stabilizing). For CH, the high voltage and frequency pair is always used.

Fig. 5.9 compares the energy consumption of the AUTO application under both energy-aware mapping (cluster 4) and random mapping (cluster 3). CDDR, DDVFS (with two sets of load boundaries) and CH are analyzed for both clusters. The same latency boundary is set both clusters (labeled in the figure). The energy consumption is normalized with the maximal per-stream energy (random mapping with CH). The per-stream latency is normalized with the latency boundary. The normalized figures are chosen for two purposes. For one thing, they emphasize the relative comparison between different approaches, with the maximum energy consumption and upper latency boundary as the units. For another, the system-level energy modeling as in Section 3.3.1 does not necessarily manifest the accurate energy consumption. In Section 7.4, RTL analysis will provide absolute energy values for implemented self-adaptive NoCs.

From Fig. 5.9, firstly the energy efficiency of CDDR, DDVFS and CH can be compared. The constant setting, CH, has the highest energy consumption as the voltage and frequency are always chosen the higher pair, while the latency is much below the boundary. DDVFS, on the other hand, reduces the energy consumption significantly (56.9% to 78.0% of the corresponding CH setting). However, as the reconfiguration is exclusively based on the local information, the global performance is not ensured. As illustrated in Fig. 5.9, with (0.15,0.25) as the load boundary, the latency boundary would be met in the random mapping case. But the same load boundary will lead to latency boundary violation, which does not satisfy the design constraint. CDDR, as considering both the cluster-level performance and local optimization, provides a proper tradeoff between energy and performance. As can be seen from



(a) Energy Consumption



(b) Stream Latency

Figure 5.9: Energy Consumption and Stream Latency of AUTO Application

Fig. 5.9 (b), the latency is initially very low with the high voltage/frequency setting for all routers. With monitoring the run-time per-stream latency, least-used routers are configured with the low voltage/frequency setting. Hence the latency is gradually increasing but flattens before reaching the threshold. In this way, the energy consumption is reduced (Fig. 5.9 (a)) with the stabilized per-stream energy being 88.4% of the CH setting in the energy-aware mapping case. The energy saving is dependent on the latency boundary. For instance, in the random mapping case, the latency boundary is reached after few iterations, thus the energy saving is minimal (4.6%). But importantly, the latency boundary is met in both cases while the energy is reduced.

In addition to the benefits of cluster-level CDDR, the contribution of platform-level energy-aware mapping can also be observed from Fig. 5.9. The energy consumption in energy-aware mapping is lower than that in random mapping, under all cluster-level energy management settings (CDDR, DDVFS and CH). In particular, with CDDR, the stabilized per-stream energy in energy-aware mapping is 68.4% of that in random mapping. It should be noted that energy-aware mapping does not necessarily lead to lower communication latency. Mapping tasks on closer processors may lead to lower energy consumption, but incur more congested network traffic. This explains why the latency in DDVFS under (0.15, 0.25) load boundary with energy-aware mapping is larger than that with random mapping.

A similar trend can be observed from the simulation of CON application, as illustrated in Fig. 5.10. The constant setting CH still incurs the highest energy consumption. DDVFS with load boundary (0.15,0.25), for this application, exceeds the latency boundary for both energy-aware and random mapping cases. In contrast, for the AUTO application, it meets the latency boundary in random mapping case. This observation further demonstrates the disadvantage of applying purely distributed energy management without global performance awareness. In terms of CDDR, it still achieves reduced energy consumption while meeting the latency boundary. The stabilized energy consumption with CDDR is 78.5% (in energy-aware mapping) and 81.1% (in random mapping) of the energy under CH. Even though DDVFS has lower energy consumption than CDDR, DDVFS can not guarantee the performance requirement unless the exact load thresholds for the latency requirement is known before execution. Such requirement is most likely infeasible in practice, since the application may change at the run-time, or the performance

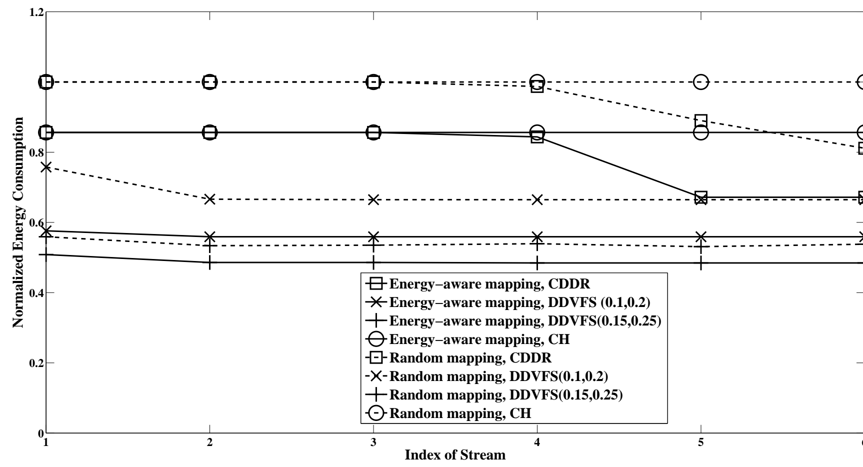
constraints can change dynamically, or the hardware platform may go through reconfiguration (e.g., in face of faults). Fig. 5.10 also shows the contribution of energy-aware mapping in the CON application. The relative energy saving varies upon the effectiveness of the random mapping. For CON application, the energy-aware mapping achieves, compared to the random mapping, 14.4% saving under CH setting, and 17.2% under CDDR.

The quantitative analysis demonstrates the effectiveness and efficiency of hierarchical agent-based energy management. The platform-level application mapping achieves large energy saving (10.0% to 37.0%), regardless of low-level techniques. Under energy-aware mapping, CDDR further reduces the energy consumption (by 11.6% to 21.6%) with distributed reconfiguration, while satisfying the performance requirment with cluster-level performance awareness. Cell agents trace the necessary local information, report to the cluster agent for cluster-level decision-making, and actuate the reconfiguration issued by the cluster agent.

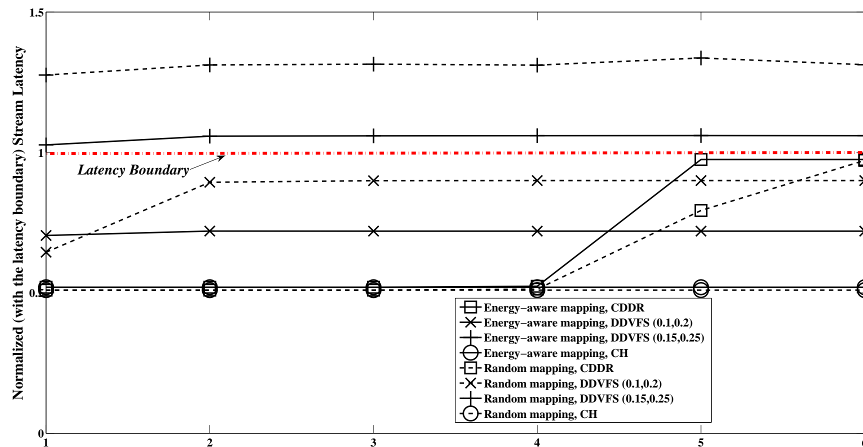
5.4.3 Implementation Discussion

The hierarchical energy management in this chapter is experimented with high-level simulation, so the complexity of agents can not be quantitatively analyzed. However, a discussion on suitable implementation alternatives for each level of agent, based on Section 4.3, can be performed.

The platform agent performs energy-aware mapping, which requires large amount of data storage (the communication volumes between tasks in all applications). In addition, depending on the exact algorithms, the data processing may be complicated. The simulated tree-model-based mapping algorithm has low timing complexity [128]. However, other potential algorithms may incur large timing complexity [78], for instance, ES(exhaustive search) or GI (greedy incremental), which cannot be eliminated from consideration on the platform agent. Thus a SW-based implementation for the mapping service on the platform agent is a proper design choice. The cluster agent, which performs CDDR (Section 5.3.3), can be either SW-based, HW-based or SW/HW co-design. As the cluster agent is tracing the load information of cells in the local cluster and the timestamps of application milestones, the amount of needed storage is quite limited. A hardware-based cluster agent can fulfill the function of CDDR, but lacks the flexibility to configure different decision-making and reconfiguration algorithms. A software-based cluster agent, on the other hand,



(a) Energy Consumption



(b) Stream Latency

Figure 5.10: Energy Consumption and Stream Latency of CON Application

incurs major area overhead (if every cluster agent is a dedicated processor) and timing overhead (if the software waits on every load information). Thus a SW/HW co-design is a proper implementation. Each cell agent, as illustrated in Fig. 5.6, integrates fine-grained counters for tracing loads and application milestones. A hardware-based implementation is most suitable to realize such modularized design.

In Chapter 7, quantitative overhead analysis of the agent structures will be elaborated based on an implemented H2A architecture.

5.5 Chapter Summary

This chapter presented hierarchical agent-based energy management following the H2A design paradigm. Based on the generic agent functional partitioning, a multi-layer energy management architecture is established, where the platform agent performs coarse-grained energy-aware resource allocation and the cluster and cell agents perform fine-grained energy-driven reconfiguration. As an example of energy-aware resource allocation, the architecture integrates tree-model-based application mapping, which maps multiple applications onto different clusters in a many-core system. For fine-grained cluster-level energy management, CDDR (clustered decision-making, distributed reconfiguration) was proposed. The reconfiguration decision made by the cluster agent is based on hybrid monitoring of the local traffic load and application milestones. By monitoring important application instructions (milestones), the cluster agent is aware of the application performance, e.g., the latency. Thus instead of oblivious reconfiguration, the cluster agent can achieve energy saving while directly tracing the performance. The distributed reconfiguration supports fine-grained energy saving exploiting the spatial locality of network traffics. In addition, the design of cell agents to trace local loads and application milestones was presented to support the cluster-level energy management.

Quantitative analysis with two practical applications was performed. The results confirmed that both energy-aware saving and CDDR contributed to the energy saving. In particular, CDDR is able to gradually reduce the energy consumption until the performance threshold is approached. Comparatively, purely distributed DVFS, while significantly saving the energy consumption, can not provide performance guarantee with only local awareness.

The hierarchical energy management in this chapter assumes static cluster

partition during execution. Thus a cell agent always reports to the same cluster agent. What if the relation between cluster and cell agents needs to be dynamically reconfigured, for instance, in different applications or in face of permanent failures? The cluster agent needs to be aware of the run-time organization of the cluster, in order to collect the information from the right cells. In the next chapter, a dynamic clusterization architecture will be proposed to address this issue.

Chapter 6

Dynamic Clusterization for Dependable Computing

The previous chapter presents hierarchical energy management, where the assignment of cells into clusters is performed before execution. However, the constant scaling of technology introduces profound variations, unpredictable errors and failures, thus the situations when certain cells fail at the run-time must be properly dealt with. This chapter proposes dynamic clusterization to support dependable computing on hierarchical agent-based systems. In particular, any cells can be dynamically organized into any cluster in case of processor failures or performance degradation. Hierarchical agent management can properly continue based on the dynamically reconfigured clusters. The chapter will present the architectural design of the supporting structures, including a Resource Look-up Table (RLT) for the platform agent, a Cluster Look-up Table (CLT) for each cluster agent, a Cluster-Identifier Register (CIR) and a Rerouting Table (RT) for each cell agent. In addition, a mesh-based monitoring network enables the communication of any cell agent to any cluster agent with minimal-distance routing.

6.1 System Architecture Overview

Dependability is a major design challenge on many-core systems. While an increasing number of resources can be integrated onto a single die, the fault occurrence is also rising [105]. For one thing, due to the small feature size, process variation and aging, the probability of permanent and transient faults

increases in VLSI systems [22]. For another, the deviation in the supply voltage and threshold voltage may lead to longer critical paths and consequent worse performance [117]. When certain resources in a many-core system fail, the system should still properly perform with the remaining resources, in order to provide dependable computing and improve the yield [105].

To support dependable computing in a hierarchical agent-based system, dynamic clusterization is proposed, which allows any cell to host the cluster agent, and any cells to be assigned to a particular cluster. A motivational example is illustrated in Fig. 6.1. While an application is running on a many-core platform, processor failures can occur unpredictably. Besides, the performance of the system can degrade in case the critical path is longer than the nominal design due to variations in the threshold voltage. To account for the unpredictable defects, providing spare cores is a widely acknowledged technique [130]. In addition, for multi-threaded applications where the threads can be dynamically loaded to more number of processors, adding spare cores may increase the performance [12]. As illustrated in Fig. 6.1, the processor running task *SINK* fails, thus processor $SP5$ is used for its replacement. Task $IFFT$ running on $P3$ and $P4$ has a lower speed than expectation, thus processors $SP3$ and $SP4$ are added to run the task. In a hierarchical agent-based system where clusters are formed to run individual applications, spare cores need to be assigned as cells and allocated to a cluster. In this example, $SP3$, $SP4$ and $SP5$ will be assigned to the cluster agent. In case a cell hosting one cluster agent fails, another cell should be configured to host the cluster agent. Dynamic clusterization is designed for such purposes.

Dynamic clusterization, as in any other self-adaptation processes, includes monitoring (M), decision-making (D) and reconfiguration (R) stages (Fig. 6.2). The monitoring stage identifies processor failures or variation-induced timing errors (leading to performance degradation). The detection of processor failures can be performed in a variety of manners, for instance, by running test programs on a processor to compare the outputs [91]. In addition, the timing errors induced by the process variations can also be detected by comparing the register output with a delayed latch output [27]. Such run-time testing can be issued by the platform, cluster or cell agents, for instance, by temporarily taking a processor from normal operation and running the test program. As the testing techniques for processor failures are beyond the scope of this thesis, it is assumed that cell agents can detect processor failures and

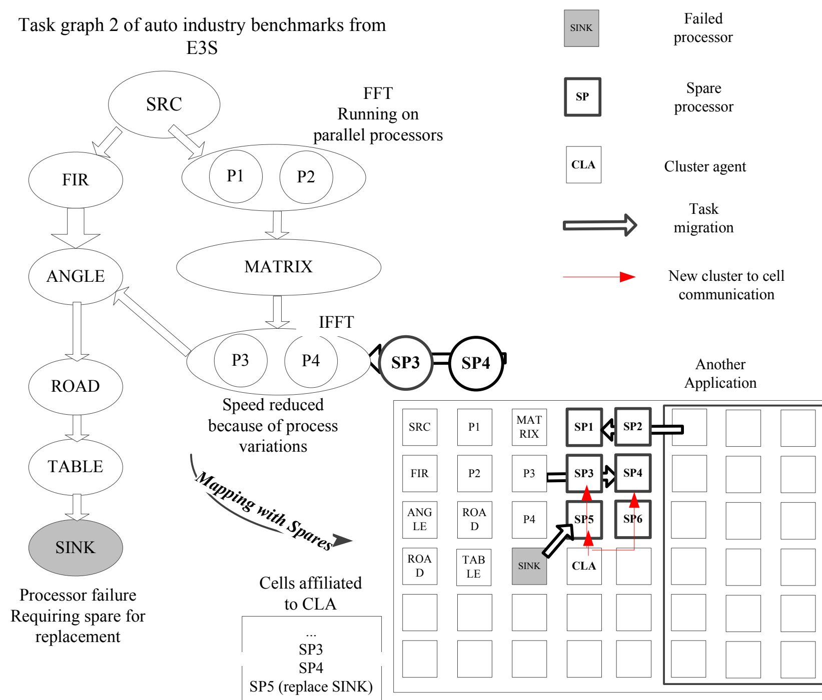


Figure 6.1: Dynamic Clusterization for Fault-Tolerance and Performance Degradation

notify their occurrences to the platform agent. The fault status of all cells is stored in the platform agent, which determines the proper spares to be assigned to each involved cluster. If necessary, a complete remapping of tasks onto the new cluster can be performed by the platform agent, for instance, to minimize the communication energy considering the utilized spares. The spare replacement and remapping decisions, as made by the platform agent, will be actuated in the reconfiguration stage. In particular, spares for replacement will be notified of the cluster agent's location. Similarly, the cluster that receives new cells will be updated with the cells' locations. The reconfiguration not only applies to spares, but also to cells that are reconfigured to a different cluster. In case a system is short of resources accommodating all applications, the platform agent may decide to reallocate a processor from a lower-prioritized cluster (based on its application) to a cluster running a more critical task.

6.2 Three-Level Supporting Structures

On each level of agents, there is a specific structure supporting dynamic clusterization. Resource Look-up Table (RLT) is a reconfigurable storage of all cells' monitored parameters, which are accessible to the platform agent. As illustrated by Fig. 6.3, RLT has the number of entries as large as the total number of cells in the system. In each entry, fields related to the cell's utilization status (used or spare) and fault status (proper or broken). The utilization and fault status are needed for the platform agent to determine the mapping, for instance, choosing the closest replacement for a broken core. The entries in RLT are updated by the platform agent, either upon receiving new cell status from the cell agents, or after the platform agent decides on a new clusterization.

For each cluster agent, there is a Cluster Look-up Table (CLT), which records the status of cells currently allocated to the cluster, and the information of application(s) currently running in the cluster. As illustrated by Fig. 6.4, the number of entries in CLT is the number of cells and applications allocated to the cluster. Any broken cell will be cleared by removing its entry in CLT. Each entry for cells include the cell ID (the address of the cell in the network), and the run-time parameters (usually design specific such as buffer load, cache hit rate). The run-time parameters are used for adaptive

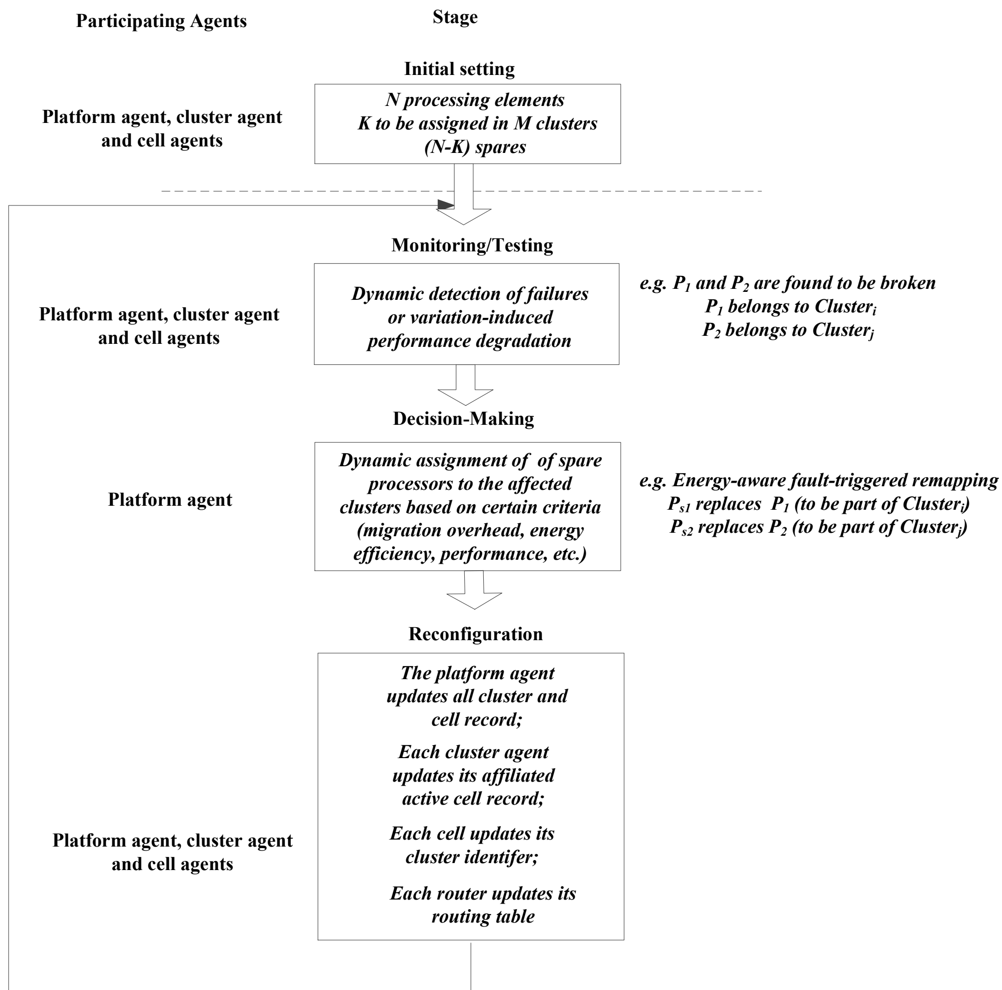


Figure 6.2: Monitoring, Decision-Making and Reconfiguration Stages in Dynamic Clusterization

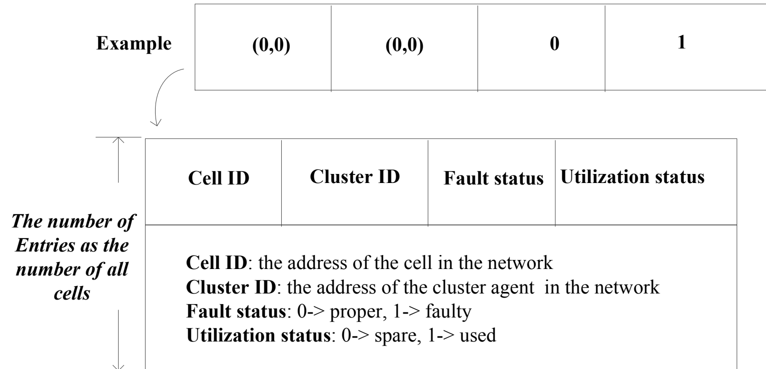


Figure 6.3: RLT for Resource Tracing on Platform Agent

optimization. For instance, the buffer load can indicate network congestion in energy-performance trade-off (Chapter 5). Each entry for applications include the application ID and timestamps of monitored milestones (e.g., the starting/end time). Compared to the content in RLT, CLT does not include the fields of cell utilization and fault status, as all cells stored in CLT are actually being utilized. When new cells are allocated to a cluster, or broken cells are removed from the cluster, the platform agent will update the CLT accordingly. On the other hand, the run-time optimization (such as the energy management) is done at the cluster level, thus the cell and application parameters, as reported by each cell agent, are stored only in the CLT.

On each cell agent, there is a cluster identifier register (CIR), which stores the location of the cluster agent. As a cell (utilized or spare) can be allocated to any cluster at the run-time, CIR, being reconfigurable by the platform agent, is written with the current cluster agent's address. In addition, to support the data communication after fault-triggered remapping, a re-routing table (RT) is attached to each router, which is also written by the platform agent. When a broken processor is replaced by a new processor, all packets destined for the broken processor will be modified with a new destination (in the header flit).

6.3 Inter-Agent Monitoring Network

The platform agent, cluster agents and cell agents communicate to send status information and configuration commands, during the adaptive services. The monitoring communication is important to the system awareness. In case the

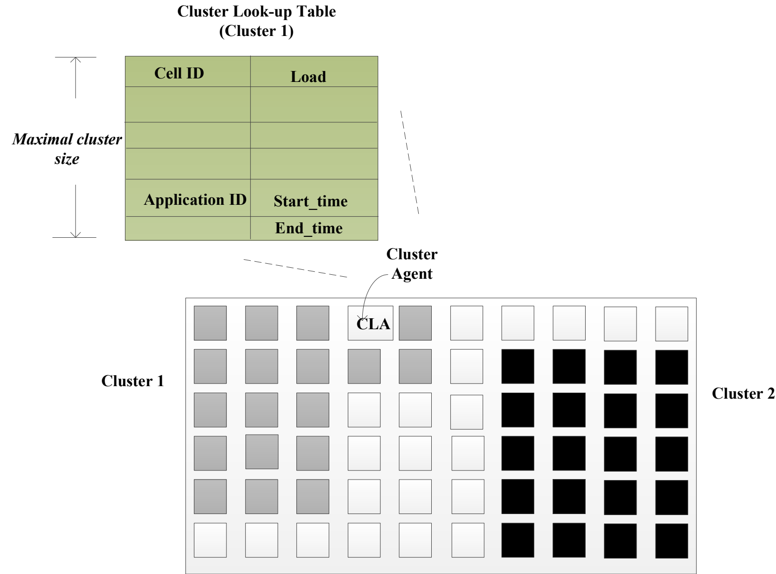


Figure 6.4: CLT for Currently Utilized Cells in Each Cluster

monitoring communication is delayed or lost, neither the resources' nor the agents' status is visible. In particular, the dynamic clusterization allows for any network node to host a cluster agent, and cells can be assigned to any cluster. Thus, a monitoring network needs to be built to support such communication. The design of the network should fulfill the following objectives:

1. Guaranteed Services. In terms of network communication, there are generally two types of Quality-of-Service (QoS): best-effort and guaranteed communication. Best-effort service, which provides no special treatment to individual traffic classes, has the lowest design complexity, but is not able to offer predictable and guaranteed performance. Guaranteed services, on the other hand, ensure certain metrics of performance (e.g., guaranteed bandwidth or bounded latency) for a specific traffic class. To ensure the system's visibility, the monitoring communication needs to be properly transmitted regardless of the status of the data communication. In particular, the network should support predictable and guaranteed latency to the monitoring communication.

2. At least one path between every two nodes. As required in the dynamic clusterization, any cell can be allocated to a specific cluster. Since a cluster agent can be hosted on any network node, connectivity between every

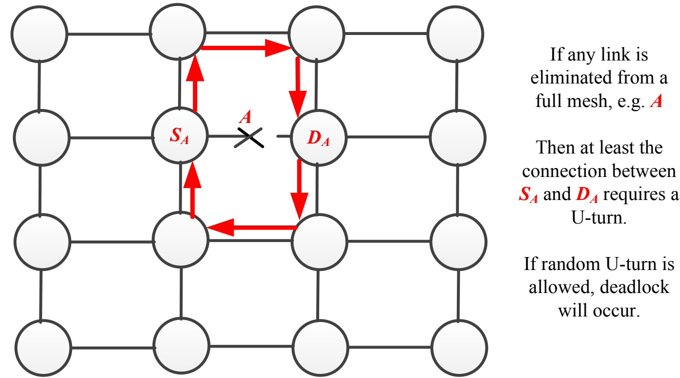


Figure 6.5: Mesh Network for Minimum-Distance Dimension-Order Routing

two nodes is required to allow the maximal clusterization flexibility. Assuming a 2-D mesh-based NoC for data communication, a mesh-based monitoring network in parallel to the data channel supports such connectivity with the most simple dimension-order routing algorithms. The elimination of any channel in the mesh network would require a more complicated routing algorithm to provide minimum-distance routing (no U-turns) and deadlock freedom(Fig. 6.5).

3. Energy Efficiency. Despite the relatively low volume of monitoring traffic compared to the data traffic, the energy efficiency is still a highly prioritized requirement. With more fine-grained monitoring and reconfiguration operations in massively parallel on-chip systems, the traffic volume will increase. In addition, for ultra-low-power applications (e.g., wireless sensor networks), the monitoring flow will be the major source of power consumption when the data traffic becomes very low, since the monitoring services cannot be completely turned off.

4. Affordable Area Overhead. This conventional design constraint is alleviated in current and emerging on-chip systems, since quite abundant wirings can be provided by the state-of-the-art multi-layer fabrication process [124]. Nonetheless, the area overhead should still be made small and scalable.

Several interconnection alternatives can be utilized for the monitoring communication, including the following architectures:

- Baseline best-effort architecture. With the agent communication overlapped with data communication without special treatment (Fig. 6.6 (a)), this alternative is considered as a best-effort interconnection architecture, used as the baseline for comparison with guaranteed services. The best-effort architecture suffers from performance disadvantages. For one thing, the monitoring communication is coupled with data traffic, which leads to unpredictable latency. When the network faces heavy traffic load, the latency increases significantly and the agent communication will suffer from similar delays. In addition, the monitoring communication has to share the same speed as the data communication. For urgent information, the monitoring communication can not be flexibly reconfigured with high speed. The benefit of this baseline architecture is low router complexity and reduced area overhead for wiring.

- Virtual channel based multi-accessing. With virtual channels, guaranteed services can be provided to specific traffic. TDMA (Time Division Multiple Access) is a widely used multi-accessing technique [73], where timeslots can be reserved for the monitoring communication (Fig. 6.6 (b)). Buffers are allocated to data and monitoring communication separately, in order to decouple the two traffic classes with virtual channels. TDMA-based connection provides guaranteed bandwidth with moderate reconfigurability. The silicon area is increased compared to the baseline alternative as routers need to integrate two virtual channels. More importantly, virtual channeling increases the energy consumption significantly for every traversal in the switching fabric.

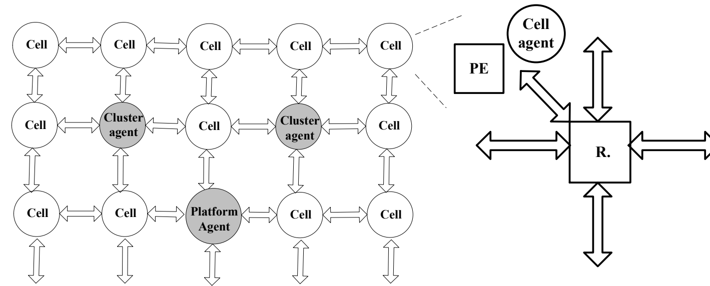
- Physically separate network. Physically decoupling the agent monitoring communication from data traffic results in a physically separate monitoring network (Fig. 6.6 (c)). As the dynamic clusterization allows any cell to be allocated to a cluster agent, the monitoring network requires a separate router on every NoC node (Fig. 6.6 (c). This alternative provides guaranteed bandwidth to the monitoring communication. Thus the latency of monitoring communication is predictable and bounded, as its volume is predictable and low. Compared to the virtual channel based networks, this architecture significantly reduces the switching and arbitration complexity in the switching fabric, which provides high energy efficiency while costing more wiring overhead. In addition, it supports

the flexible configuration of the monitoring network (e.g., its switching frequency) independently of the data communication.

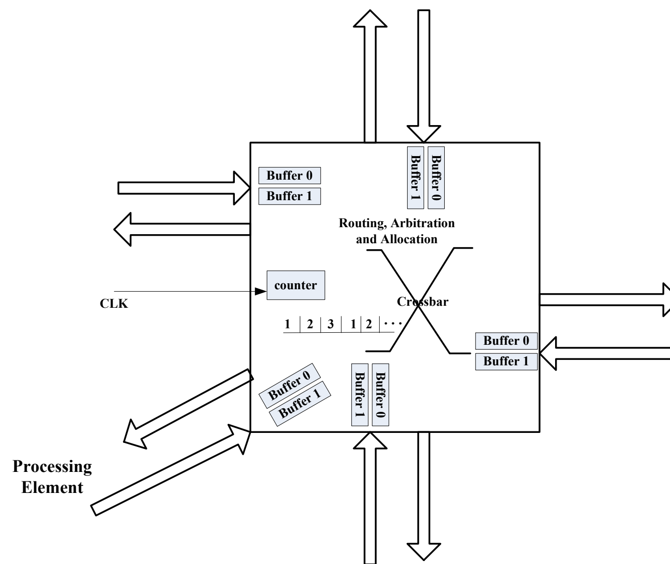
The three alternatives are not exhaustive, as other interconnection architectures are also potentially useful. For instance, priority-based switching for monitoring packets can be applied when the monitoring and data communication are overlapped on the same physical channels. However, such technique can lead to blocking of monitoring packets when the data communication occupies the channel buffers. Exception can be made for deflection routing [26], where the packets can be routed to any outputs when the desired output is occupied. An example will be given in Section 7.1.

To achieve quantitative analysis of the energy-performance trade-off, simulations are performed with the same platform and traffic setting as in Section 3.3. In brief, four traffic traces (B,H), (B,U), (U,U), (U,H) (B: b-model, H: hotspot, U: uniform) are experimented with the same setting as Fig. 3.10, except for the last traffic (U,H). The injection rate for (U,H) is increased from 2.24Gbps/node to 3.2Gbps/node, in order to demonstrate the influence of intensive data communication on the monitoring latency. Without implying any restrictions, each cell agent sends 1 packet (8 flits) to the cluster agent every 100 cycles. Each data channel is 32-bit wide and 1mm long. For physically separate monitoring channels, they are 8-bit wide and also 1mm long. All routers work at the same voltage and frequency (V_H=2V,F_H=1GHz), to eliminate the influence of dynamic power management. For the TDMA-based architecture, one out of three timeslots gives priority to the monitoring communication. Only when there is no monitoring traffic will the timeslot be used for data communication. Fig. 6.7 illustrates the average latency and energy consumption of the monitoring communication with the three architectural alternatives.

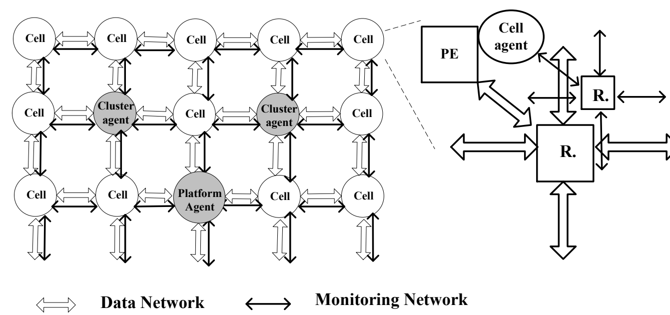
From Fig. 6.7, it can be observed that the monitoring communication decoupled from the data traffic provides guaranteed latencies and saves the energy consumption. The physically separate monitoring network returns the same latency and energy consumption independently of the data traffic. TDMA also provides guaranteed bandwidth, but its latency and energy consumption are much higher than the physically separate network (197.22% and 60.19% respectively). The higher latency is due to the lower switching frequency (only 1 in 3 slots is used for monitoring communication). The energy consumption is increased due to the wider data width (32-bit vs. 8-bit) and



(a) Baseline Interconnection Architecture

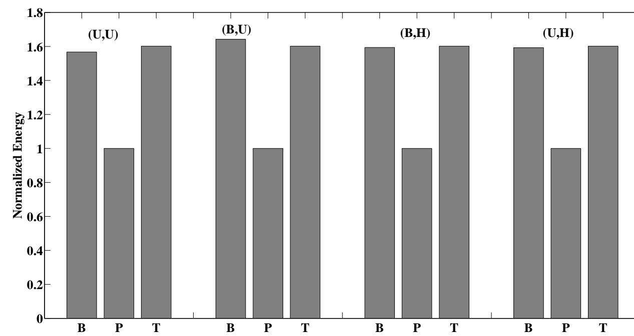


(b) TDMA-based Router

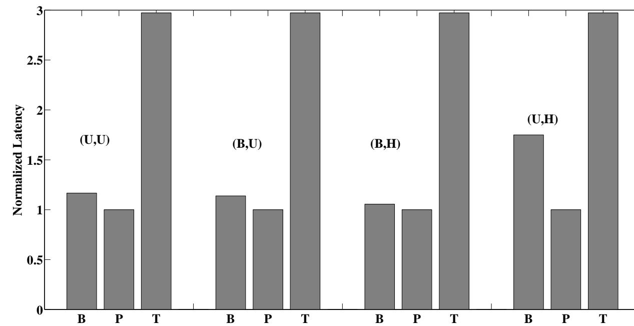


(c) Physically Separate Monitoring Network

Figure 6.6: Interconnection Architectures for Inter-Agent Monitoring Communication



(a) Normalized Energy Consumption



(b) Normalized Per-Flit Latency

Figure 6.7: Energy Consumption and Latency of Three Monitoring Communication Architectures (B: baseline, P: physically separate monitoring network, T: TDMA-based router architecture; (B,H), (B,U), (U,U), (U,H)): the four data traffic patterns.

Table 6.1: Area Estimation (um^2) for Each Interconnection Architecture

Architecture	Data Router	Monitor Router	+	Data Wire	Monitor Wire	+
Baseline	111969	0	0	130000	0	0
TDMA	114960	0	2.7%	130000	0	0
Separate Monitoring Network	111969	8135	7.3%	130000	34000	26.2%

virtual channels in the routers. The baseline architecture, in addition, cannot guarantee the latency. With a high injection rate in (U,H), the monitoring communication also faces long latency (75% longer than the physically separate network). The baseline architecture also has higher energy consumption than the physically separate network (56.74% - 64.23% higher), due to the wider data width in the routers.

The area overheads of the three alternatives can be estimated by Orion 2.0, divided into router and wiring areas, as illustrated in Table 6.1. It can be identified that the physically separate network adds 7.3% router area overhead (mostly as transistor area) and 26.2% wire area overhead, compared to the baseline architecture. The transistor area overhead is quite small. Even though the metal wire consumption is higher than that in the baseline architecture, such overhead is feasible given the constant technology progress in multi-layer chip fabrication (see Section 3.1.4 for more reasoning). Based on the trade-off between energy consumption, latency and area overhead, the physically separate monitoring network best fits the design criteria with guaranteed performance, low energy consumption and affordable silicon area.

6.4 System Integration

To support the dynamic clusterization, RLT, CLT and CIR need to be integrated on three levels of agents, with a physically separate monitoring network. The dynamic clusterization process handling processor failures is illustrated in Fig. 6.8. Stream applications are assumed. At the run-time, before loading a new stream, all cell agents perform a self-test. Any processor failure will be notified to the platform agent. Upon receiving the fault status, the platform

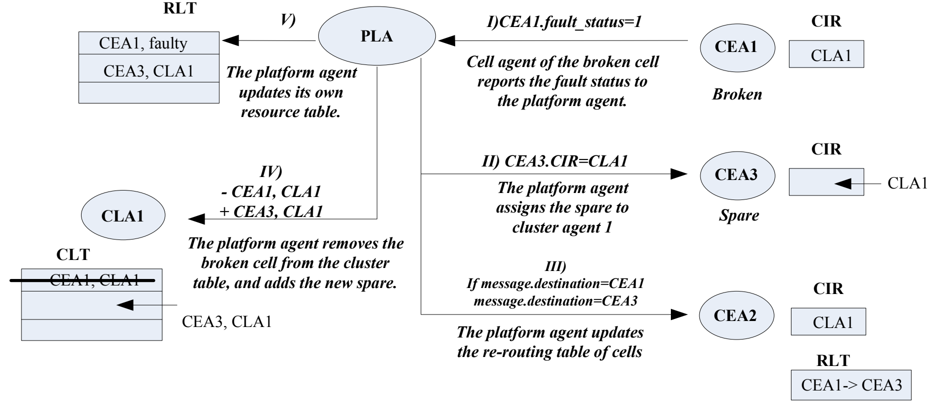


Figure 6.8: Inter-Agent Communication Process for Dynamic Clusterization

agent calculates fault-triggered remapping, which attempts to locate the most suitable spare cell for the broken cell. As the platform agent is aware of the processor status in all clusters, it can assign the spare cells without locating the same spare in two clusters. Based on the remapping decisions, the platform agent continues with dynamic clusterization. It writes the CIR of the chosen spare cell with the address of the cluster agent. The RT of each router is also updated by the platform agent, so that the spare cell's address will replace that of the broken cell. For the cluster agent, the platform agent updates its CLT by removing the broken cell and adding the new cell. Lastly, the platform agent updates its own RLT, based on the new cluster information. Then a new stream arrives on the properly configured platform.

The look-up table structures can be implemented either by software or hardware. In particular, RLT is a relatively large piece of storage, whose number of entries is equal to the number of cells in the whole network (Fig. 6.3). Such a look-up table can be realized in the local memory of the platform agent, with the overhead incurred only once in the system. CLT is a smaller piece of storage, whose number of entries is equal to the maximum number of cells in a cluster plus the application milestones (Fig. 6.4). Even though the theoretical boundary of the cluster size is the platform size, a typical cluster size should be configured much smaller to just accommodate an application. Otherwise, the benefits of fine-granularity control no longer exist. Consequently, a CLT can be implemented as dedicated hardware. Not only does it allow flexible pre-processing on the CLT, but also any cell can host a cluster

agent by adding a small CLT. At the design time, the CLT can be properly sized as the maximal application size and application milestone number are known before execution. CIR is a very small register within each cell agent. RT can be integrated into any routing table present in the NoC platform (as will be implemented in Section 7).

6.5 Further Discussion

The dynamic clusterization presented in this chapter assumes processor failures as the fault model. Addressing the link errors will expand the fault tolerance of the NoC platform, as partially discussed in a related work [3]. If there are spare wires, the connectivity can still be maintained with the same dynamic clusterization process. In case a link is permanently disconnected, adaptive rerouting may be needed for data and monitoring communication. In addition, the remapping algorithm should consider the inaccessibility of disconnected cells. This chapter focuses on the architectural support of the hierarchical organization, so that a broken cell does not interrupt the function of the involved cluster and the whole platform. The problems of recovering from other types of errors or failures are beyond the scope of the thesis.

The configuration of the NoC platform during the dynamic clusterization process has implementation-dependent details. For one thing, the remapping algorithms can aim at different cost functions such as minimum task migration or energy consumption. The H2A design paradigm does not impose any constraints on the remapping algorithms, which are determined by the application requirements and system constraints (e.g., upon the algorithm complexity). Chapter 7 will utilize a tree-model-based energy-aware remapping algorithm as an example. In addition, depending on the memory architecture, remapping may require reloading of instructions onto processors, or reconfiguration of instruction addresses in case of shared instruction memory. Implementation examples will be given in Chapter 7.

6.6 Chapter Summary

Hierarchical agent-based systems face unpredictable errors and failures. To improve the system dependability, dynamic clusterization was proposed, which allows any cell to be allocated to any cluster at the run-time. In particular, a

Resource Look-up Table (RLT), a Cluster Look-up Table (CLT) and a Cluster Identifier Register (CIR) were designed to update the cluster configuration when failures occur. A dynamic clusterization process was presented to maintain the hierarchical organization in case of processor failures. In addition, a physically separate monitoring network was designed and demonstrated to be an appropriate trade-off between configuration complexity, energy overhead and area overhead, while supporting the dynamic clusterization.

Importantly, the presented supporting structures (RLT, CLT, CIR and the monitoring network) can be integrated for other adaptive services, for instance, the hierarchical energy management (Chapter 6). The entries in the look-up tables include the run-time parameters of the clusters and cells, which are utilized in the energy management and other potential services. Such a generic architecture is what the thesis aims for in the proposition of the H2A paradigm. The next chapter will implement a Self-Aware and Adaptive NoC (SAA-NoC), where the dynamic clusterization and hierarchical energy management will be realized on the register transfer level, so that quantitative verification and analysis can be achieved.

Chapter 7

Implementation of Self-Aware and Adaptive NoCs

While the previous chapters have presented the concept and design of H2A paradigm, in particular for energy management and dependability services, a concrete implementation- Self-Aware and Adaptive Network-on-Chip (SAA-NoC), is elaborated in this chapter as a proof-of-concept of self-aware and adaptive systems. It will present the microarchitectural-level implementation of the platform, cluster and cell agents, in addition to the supporting structures for self-adaptation on each level. The implementation is programmed on a RTL NoC experimental platform with Leon 3 processor, Nostrum communication architecture and distributed shared memory. The platform agent is implemented as high-level software functions. The cluster agent is implemented as SW/HW co-synthesis including a hardware-based CLT (cluster look-up table) and software functions running on one processor. The cell agent is wrapped within each NoC node as hardware circuits. Both hierarchical energy management and dynamic clusterization services will be experimented, with two practical applications (one of image processing and another for MPEG encoding). The results demonstrate the feasibility, effectiveness and efficiency of H2A mapped on a NoC, the mainstream high-performance parallel computing platform. In addition, the quantitative overhead analysis shows that both software and hardware overheads of agents and the supporting structures are minimal and scalable with the number of processors in the NoC.

7.1 NoC Platform

The SAA-NoC is implemented upon a NoC platform with Nostrum communication architecture [52], Leon 3 processors and distributed shared on-chip memory [17], as illustrated in Fig. 7.1.

Nostrum NoC [52] is a general-purpose communication architecture for many-core systems. It is a mesh-based direct network, where each node contains a router, a processing element and memory. Nostrum adopts a highly-regular physical layout. The distance between every two routers is the same, so that the electric properties (e.g., the timing) of inter-router communication are uniform. Each router utilizes bufferless wormhole switching. The router utilizes X-Y deterministic routing in case of no congestion, otherwise uses hot-potato deflection routing [87]. With hot-potato routing, the data transmission supports no dropping in bufferless flow control. Flits that cannot be transmitted to the desired output (e.g., due to contention) are simply deflected. The arbitration always picks the flit with the highest hop count in case of output conflicts [43], thus the routing is livelock free. The major features of Nostrum NoC are the regularity of the physical layout, which provides uniform and predictable electrical properties, and the simple router architecture, which reduces the area and energy consumption [92].

Leon 3 [32] is a general-purpose 32-bit processor, written in synthesizable VHDL. It implements SPARC V8 instruction set. Leon 3 processor support a total of 15 asynchronous interrupts.

In addition to a Nostrum router and a Leon 3 processor, each NoC node has a distributed memory with shared memory space [17] on all processors. In particular, each processor has $1MB$ private memory and $64KB$ shared memory [18]. A distributed memory controller is attached to each node for the management of remote and local memory accesses. The memory management (e.g., the virtual to physical address translation) is reconfigurable by loading different instructions in the instruction memory ($16KB$) of the controller. The memory controller is small ($51K$ gates) and fast ($455MHz$ in $130nm$).

To allow for each router or processor running on a different frequency and voltage, the NoC platform supports GRLS (globally ratiochronous locally synchronous) timing [13, 14]. Ratiochronous timing requires that any local clock should be rationally related to a reference clock ($f_l = \frac{1}{N} f_r$, N is an integer). It simplifies the synchronization overhead of GALS (globally asyn-

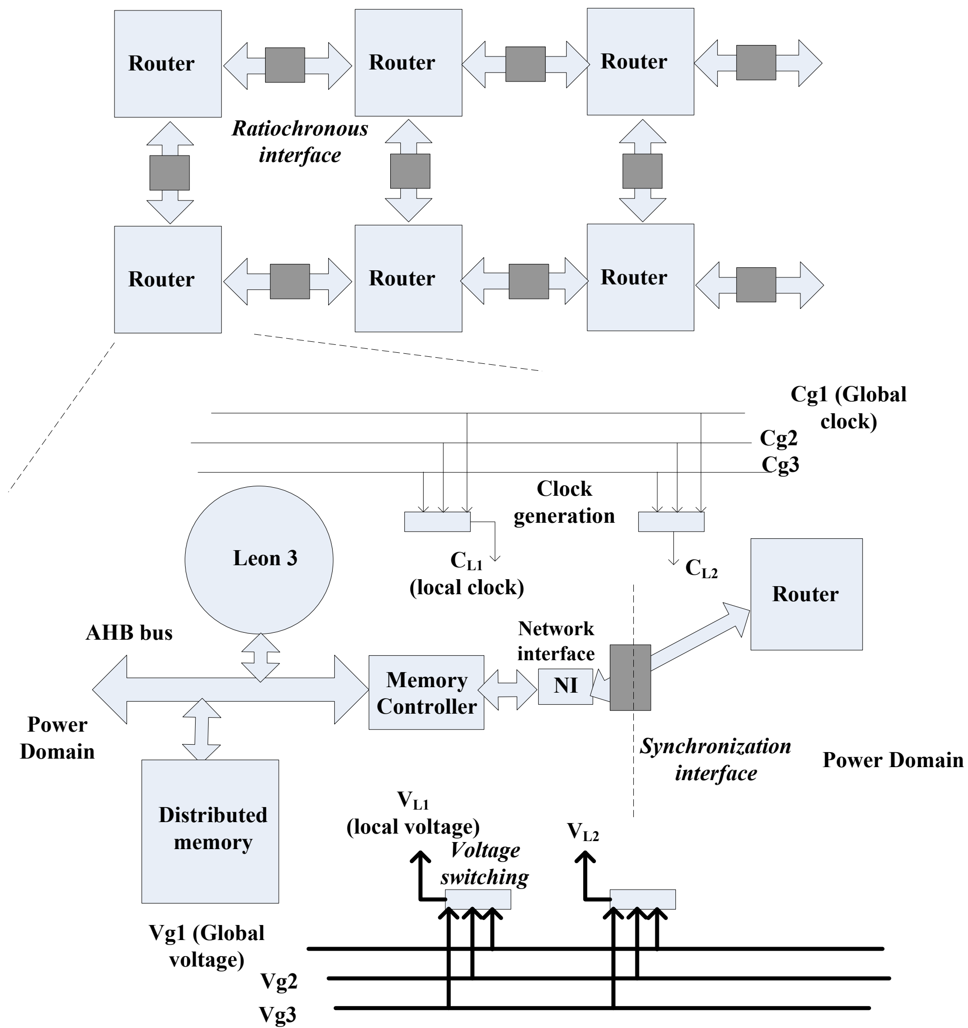


Figure 7.1: Distributed Shared Memory NoC Platform with Fine-Grained Voltage and Clock Generation

chronous locally synchronous timing), where the timings between local clocks are not related. It has a low overhead of 4 flipflops per dataline [14]. Compared to mesochronous timing (same frequency but different phases [119]), it offers greater flexibility in adjusting the frequency of individual node, while maintaining similar overheads [13].

In summary, the underlying NoC platform is a mesh-based regular-layout network, with full integration of processors, routers and distributed shared memory. In each node, the processor and the memory are connected to the corresponding router via an AHB bus (Advanced High-performance Bus, a protocol introduced by ARM company). The memory controller handles the memory reading and writing between the network and the local memory. Each router or the processor is a synchronous unit, thus synchronization interfaces are inserted for GRLS timing. Each synchronization unit has a clock generation module and a voltage switching module. Four global clocks are distributed throughout the NoC, and the local generation module selects one among the global clocks and divides its frequency by an integer value to produce the local clock [15]. The frequency division is implemented with LFSR (linear feedback shifter register). The voltage switching module emulates the behaviors of power switches connecting to multiple on-chip power delivery networks [115]; Section 3.2.1). The voltage switching assumes a delay of $27ns$ based on SPICE simulation [15]. During voltage and frequency switching, the local synchronization unit is paused by clock gating. The GRLS timing implementation is presented by the McNoC architecture [15].

7.2 Three-Level Agents and Supporting Structures

The SAA-NoC integrates all the three-level agents and the supporting structures as proposed in Chapters 4, 5 and 6. The agents can be implemented in different SW/HW configurations. This section presents, upon the aforementioned NoC platform, the platform agent running on a dedicated software, the cluster agent running as a thread on a processor, and the cell agents as hardware attached to each NoC node. The implementation is an example for SW/HW co-design of agents with the trade-off between functionality and overheads.

7.2.1 Platform Agents

For adaptive energy management, the platform agent performs energy-aware mapping (Section 5.2). To tolerate processor failures, it utilizes fault-triggered remapping and dynamic clusterization (Chapter 6). To enable the mapping process, communication volumes between tasks are stored as metadata of the application. In case of permanent failures of processors, the platform agent chooses a spare to replace the broken processor considering the energy consumption (to be elaborated in Section 7.4.3). Then clusters will be dynamically updated accordingly. The platform agent is implemented in software on a general-purpose Leon 3 processor (Fig. 7.2). The software implementation is based on two considerations: 1) the platform agent will have diverse functions dependent on the application and system requirements, for instance, different mapping or resource allocation algorithms; 2) the operations on the platform agent are very infrequently performed and do not require fast response (compared to local reconfiguration), such as the mapping and dynamic clusterization.

As illustrated in Fig. 7.2, the Resource Look-up Table (RLT), where utilization status of all cells is stored (Section 6.2), is implemented in the distributed memory of the platform agent. Each entry of the RLT contains the location of the cell, its status and its cluster agent location(if assigned to one). The platform agent can access the RLT as normal memory location. Cell agents' report of status will be stored in the corresponding entry in the RLT. The implementation of RLT in the existing memory space instead of a dedicated hardware unit is based on two-fold considerations:

- The size of RLT can easily fit into a distributed memory space. The entry width of RLT is $\log_2 N + 2 + \log_2 N$ in bits, as the status field requires 2 bits (3 states: faulty, used, spare) and cell/cluster location each requires $\log_2 N$ bits (N is the maximal number of nodes in NoC). Given 1000 nodes in NoC, each entry becomes 22-bit wide covering 3 bytes. In this case, RLT can fit in $3KB$ memory space, which is a small portion of the $64KB$ distributed memory allocated to each processor (Section 7.1).
- Implementing RLT in the distributed memory provides high flexibility for the platform agent to access the table, as memory operations. In addition, any processor can be configured as the platform agent (during

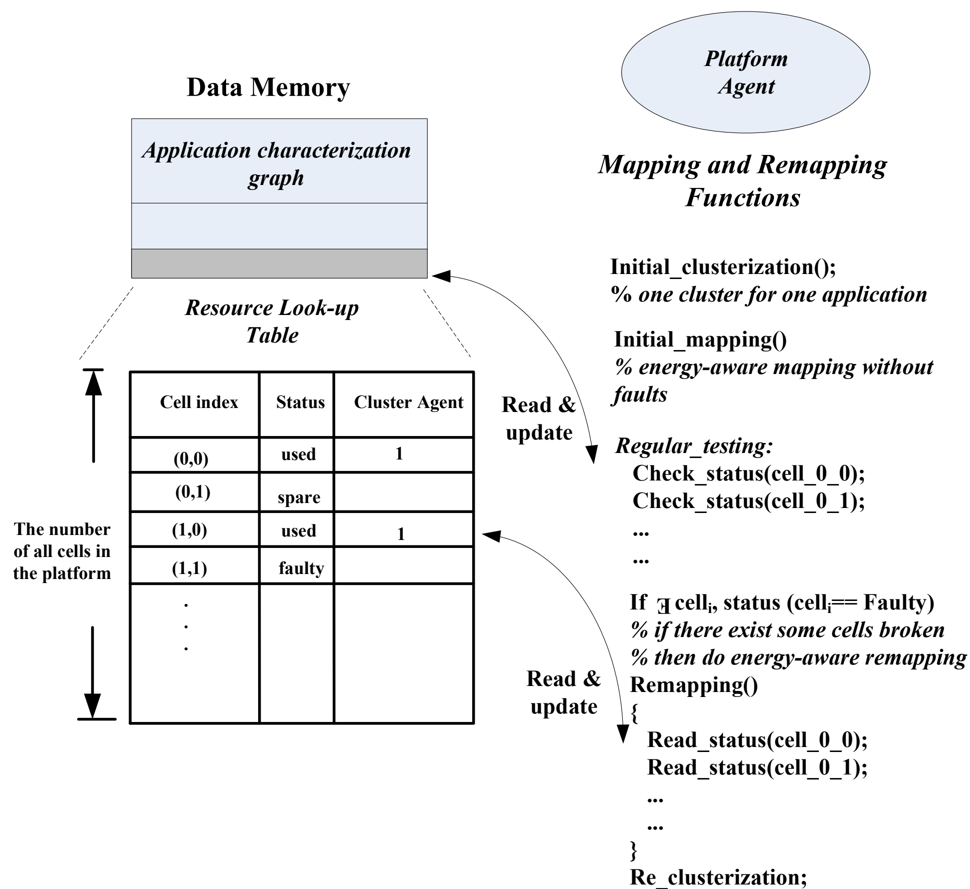


Figure 7.2: Software Implementation of Platform Agent and RLT

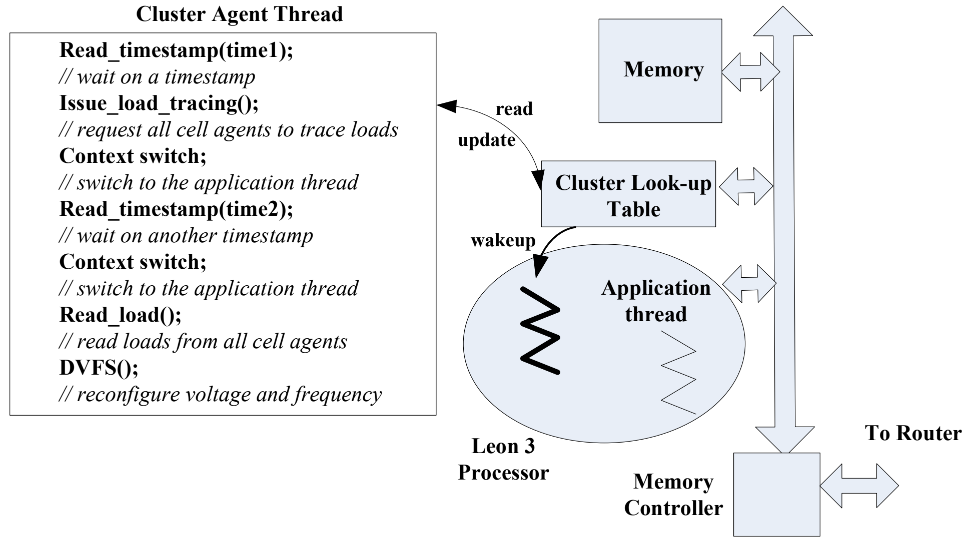


Figure 7.3: SW/HW Co-synthesis of Cluster Agent

compilation). If RLT is implemented as a dedicated hardware unit, only the cells attached with RLTs can be configured as the platform agent (unless a RLT is attached to every node).

7.2.2 Cluster Agents and CLT

Different from the platform agent, the cluster agent follows SW/HW co-design, as illustrated by Fig. 7.3. A software thread is running on a Leon processor, with input from the hardware-based Cluster Look-up Table (CLT).

The cluster agent performs cluster-based decision-making and distributed reconfiguration (CDDR). Consistent with the algorithm in Section 5.3.3, the software thread running on a Leon 3 processor provides a feedback-loop-based energy-performance trade-off. It first waits on the 1st milestone of the application, e.g., the starting instruction of a stream. When the milestone arrives, the cluster agent first notes down its timestamp and then issues load checking commands to all affiliated cell agents, requesting them to start tracing the local load. Then the cluster agent waits on the 2nd timestamp of the application (e.g., the ending instruction of the stream). When the milestone arrives, the cluster agent collects the loads (in this implementation, the peak load of a monitoring window) from all cell agents. Based on the load information and the application execution time, the cluster agent decides on the voltage and

frequency reconfiguration.

All these operations of a cluster agent run in one thread, which can be interleaved with application threads, if the processor supports multi-threading. When an instruction is suspended, for instance, when waiting on the application timestamps, the processor simply makes a context switch to the application threads. This work does not realize a full implementation of multi-threaded Leon 3 processor. Instead, we emulate such behavior with one cluster agent thread, which can be waken up by an interrupt (Fig. 7.3). Compared to a dedicated processor, running the cluster agent as a thread improves the resource utilization, while preserving the functional flexibility of software-based agents.

Each cluster agent is supported by a CLT functioning as a pre-processing unit to the processor. When the cluster agent thread waits on specific information (e.g., the local loads), the thread suspends. The information (loads or application timestamps) sent from each cell agent will be stored in the CLT. When a certain criteria is met, e.g., the loads of all routers are returned, an output signal is generated from CLT to wake up the suspended cluster agent thread. As illustrated by Fig. 7.4, each entry of CLT contains the monitored parameters of one cell or the timestamps of a running application. At the run-time, these parameters will be updated in an unpredictable order. The local load of each router, as an example, may be reported by the corresponding cell agent in a random order. The interrupt triggering criteria can be dynamically configured with RSM (row selection mask) and PSM (parameter selection mask). The RSM can be written by the cluster agent to choose the interested cells or applications. To choose the interested parameters, the PSM can be configured by the cluster agent. In case of dynamic clusterization, complete entries in CLT will be updated. For instance, the row for a broken cell will be replaced by one for the spare cell.

Implementing CLT in hardware is an alternative to the software-based implementation of a look-up table (e.g., the RLT). The design choice is based on two-fold considerations:

- The pre-processing unit only wakes up the processor when all needed information is ready, instead of triggering a context-switch every time one piece of information is received. Such technique reduces the overheads caused by excessive context switching.

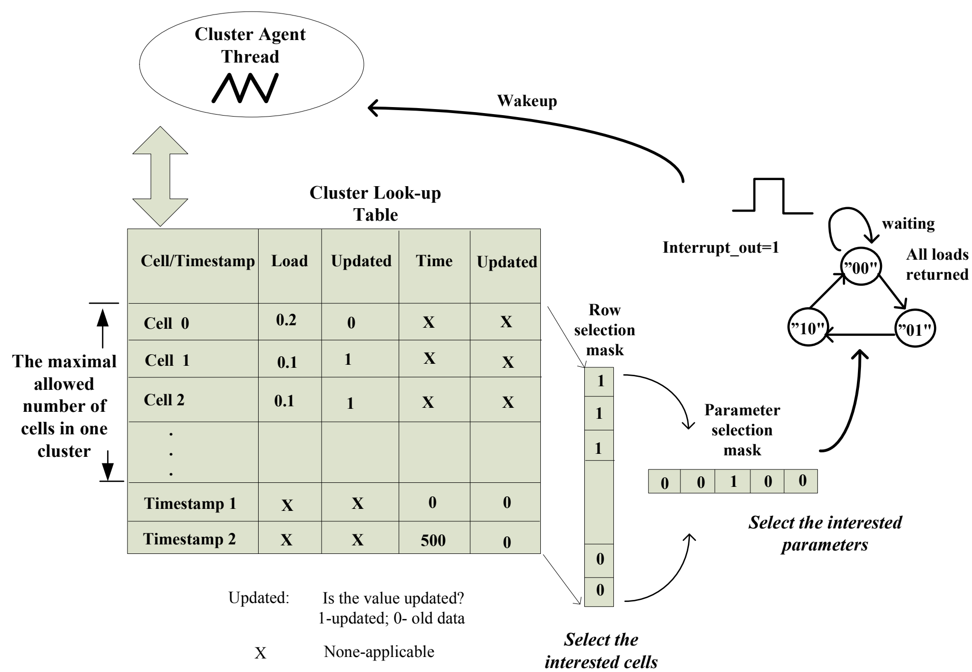


Figure 7.4: CLT: A Reconfigurable Pre-Processing Hardware Unit for Cluster Agent

- If the platform needs to support the scenario where every node can host a cluster agent (Chapter 6), one CLT should be attached to each node. Based on synthesis, a 32-entry 16-bit wide CLT only covers 9751 um^2, which is only 2.2% of a 64-bit router(more discussion in Section 7.5). Thus the hardware-based CLT implementation is highly feasible.

7.2.3 Cell Agents and CIR

Each cell agent is a hardware-based unit wrapped within a NoC node. Based on the discussion in Section 4.3 and 5.3.2, a set of local monitoring and reconfiguration operations can be modularized in every NoC node. They usually require fast response (e.g., frequency switching). Thus a modular hardware-based design is desirable where each cell agent controls local monitors and actuators (Fig. 7.5). In addition, MIR (milestone instruction register; Section 5.3.2), used for storing application milestones, is configurable by the platform agent. The CIR (cluster identifier register; Section 6.2) is also reconfigurable by the platform agent. In the dynamic clusterization process, CIR will be updated with the cluster agent address. In case of processor failures, to support remapping without recompilation, a RT (re-routing table; Section 6.2) is implemented in the memory controller, where a broken processor address is replaced with its spare processor address.

7.3 System Integration

With the three-level agents and supporting structures, the SAA-NoC implements both energy management and fault-triggered dynamic clusterization. A hierarchical view of each agent's role in these services is illustrated by Fig. 7.6, categorized by the type of services:

- For energy management: the cell agent collects the local load and traces the occurrences of application milestones, and reports them to the cluster agent. Based on the received load and calculated latency information, the cluster agent monitors the application's execution time and decides on required DVFS techniques (e.g., increasing the voltage and frequency of the most heavily-loaded routers, Fig. 5.7). The reconfiguration command will be sent to the corresponding cell agents. The platform agent reduces the energy consumption through energy-aware

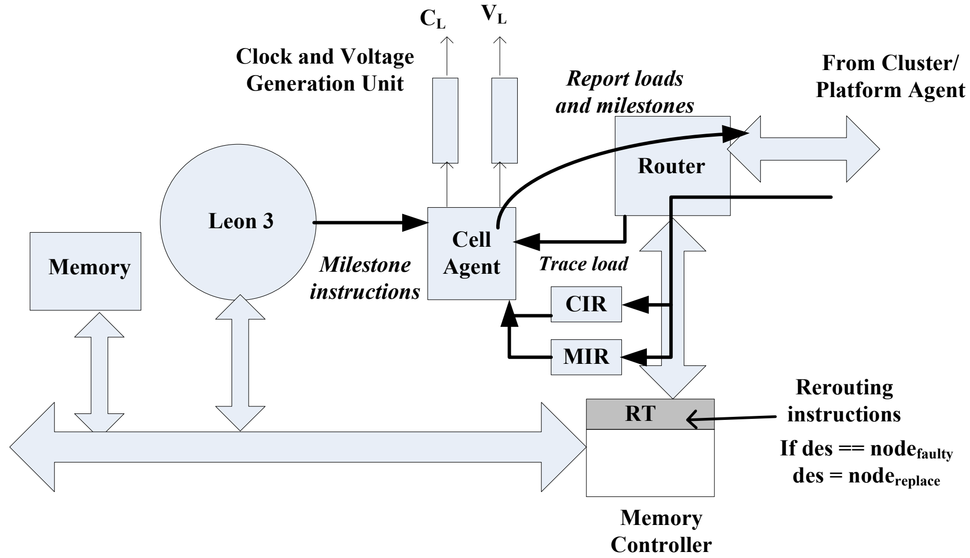


Figure 7.5: Cell Agent and the Supporting Structures wrapped in a NoC Node

mapping or remapping. The initial mapping is applied before the applications execute, when the platform decides on the most suitable task-processor assignment to minimize the communication energy (Section 5.2). In case a processor fails, the most suitable spare will be chosen based on the distances between the spare and the functional processors (to be elaborated in Section 7.4.3).

- For fault tolerance with dynamic clusterization, the cell agent reports the processor failure to the platform agent directly. Based on the fault status of cells and availability of spare cells, the platform agent decides on the proper spare for each broken cell based on the energy consumption. Afterward, dynamic clusterization will be issued by the platform agent (Chapter 6), which configures each spare to a chosen cluster. The dynamic clusterization involves the updates of RLT, CLT, CIR and RT to maintain the proper organization after spare replacement.

Upon the underlying NoC platform (Section 7.1), Fig. 7.7 illustrates the interaction between agents and supporting structures for the implemented services. Different from the physically separate monitoring network in Section 6.3, the implementation provides guaranteed monitoring communication on the data channel. The guaranteed services are achieved through two tech-

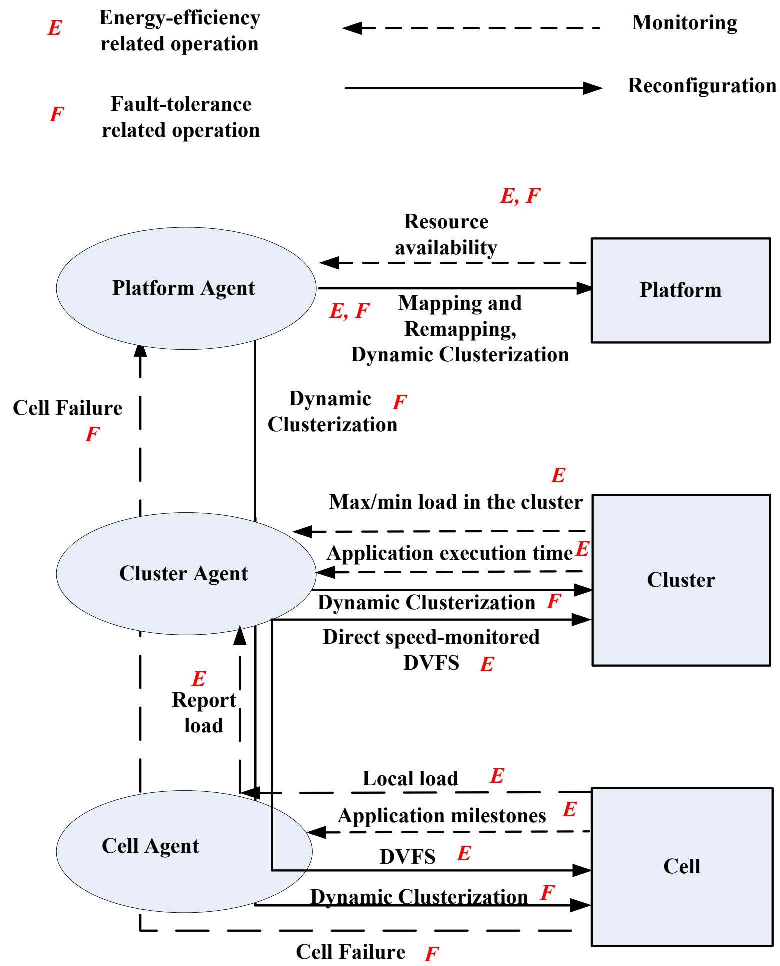


Figure 7.6: Implemented Hierarchical Agent Services

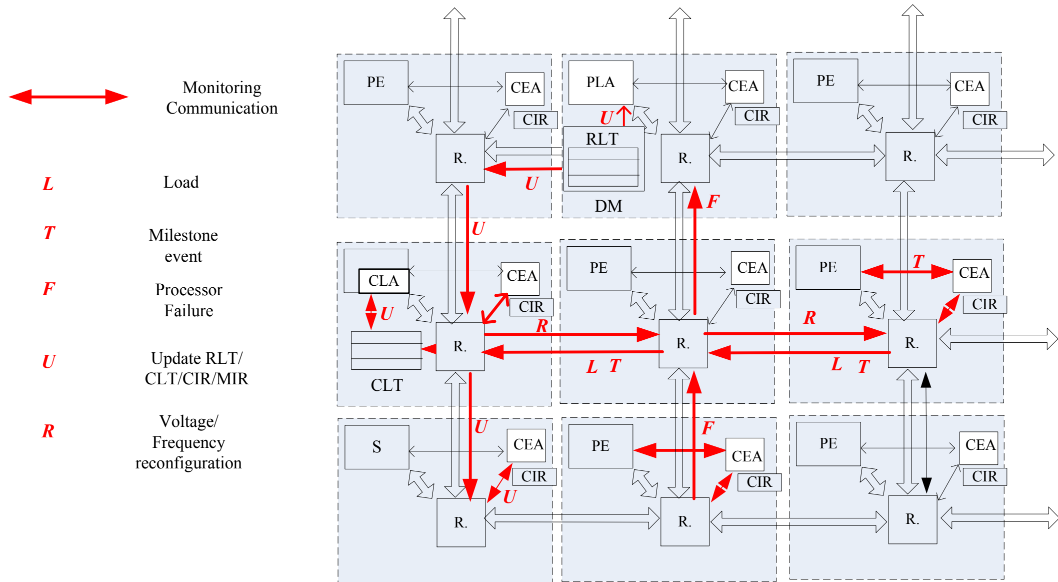


Figure 7.7: Interaction between Three-Level Agents on SAA-NoC (PLA: platform agent, CLA: cluster agent, CEA: cell agent, S: spare, DM: distributed memory, MIR is omitted for brevity)

niques: 1) the network utilizes hot-potato deflective routing (Section 7.1), thus no backward blocking will occur; 2) the arbitration process always prioritizes the monitoring communication, thus its performance is not influenced by the data communication. The guaranteed service provided by the implementation is enabled by the deflective routing, while the physically separate monitoring network in Section 6.3 is generically applicable in any routing algorithms. All monitoring communication messages are memory-mapped. Each message contains four fields (Fig. 7.8): destination, source agent, message identifier, and message data.

7.4 Experiments

With two representative applications (one for imaging process, one for video encoding), energy management and dynamic clusterization services are experimented on the SAA-NoC.

Destination	Source Agent	Message Identifier	Message Data
Memory-mapped address • CLT • RLT • Voltage, Frequency Configuration (cell agent) • CIR • RT • MIR	Node index + Agent Index (Cell/Cluster/Platform)	L: report load →	Load figure
		T: report milestone event →	Milestone instruction
		F: report failure →	{blank}
		U: update RLT/ CLT/CIR/RT/ MIR →	New entry
		R: reconfigure voltage/ frequency →	New voltage/ frequency setting

Figure 7.8: Format of Inter-Agent Monitoring Communication

7.4.1 Experimental Setting

The first application for image processing, BASIZ [96], requires 26 cores. It is mapped on a 6×5 NoC area based on its small NAD (node average distance; Section 5.2), leaving 4 cores as spares. The second application, MPEG encoding [59], requires 21 cores. It is mapped on 5×5 NoC area, also leaving 4 spare cores. As illustrated by Fig. 7.9, to analyze the respective energy saving of high-level mapping and cluster-level CDDR, one scenario with tree-based mapping (Section 5.2) and one with random mapping are experimented. To demonstrate the fault-triggered dynamic clusterization, two scenarios with process failures for each application are also included in the experiments.

To identify the voltages and their corresponding supported frequencies, the switches were synthesized using Synopsys design compiler for 65 nm multi-Vdd technology (Table 7.1). The technology supports voltages from $1.1V$ to $1.32V$. The synthesis results reveal that the routers are capable of supporting up to $300MHz$ frequency at $1.32V$ and up to $200MHz$ frequency at $1.1V$. Based on GRLS clocking in the NoC platform (Section 7.1), the allowable frequencies are 300, 100, 50, 40, and $20MHz$ (all frequencies are divisors of the largest frequency $300MHz$).

To analyze the power and energy consumption, the switching activity files are generated for each application based on the communication volume between every two tasks. Automated traffic generation is not feasible for this implementation as the application instructions are not available. The NoC

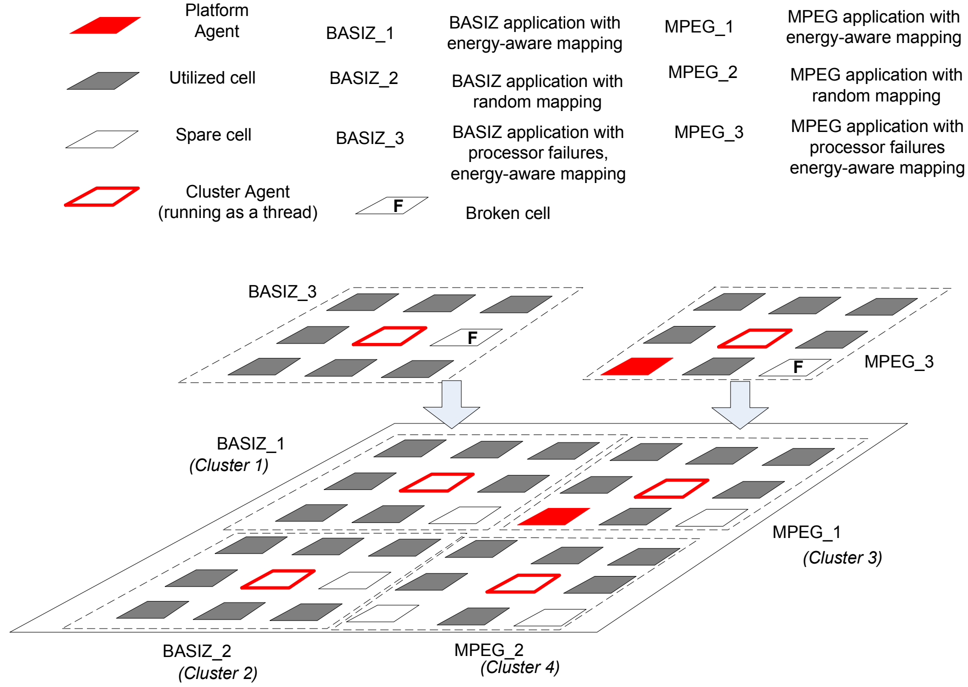


Figure 7.9: Mapping Two Applications with Six Scenarios on SAA-NoC (the number of processors is only for illustration purpose)

Table 7.1: Voltage frequency pairs (the bold font labels the allowed frequencies based on GRLS clocking and the corresponding minimal voltages meeting the timing constraints)

Voltage (V)	Frequency (MHz)	Timing constraints
1.32	400	violated
1.32	**300**	**met**
1.32	200	met
1.1	400	violated
1.1	300	violated
1.1	200	met
1.1	**100**	**met**
1.1	**50**	**met**
1.1	**20**	**met**

routers are synthesized using 65 *nm* multi-Vdd library. The power analysis is performed by Synopsys design compiler on the synthesized NoC routers with the generated switching activity files.

7.4.2 Energy Management Results

Initially all routers are configured with the highest voltage and frequency (1.32V, 300MHz). After the applications start running, each cell agent reports the local router load to the cluster agent. Two types of energy management is experimented on the four clusters (Fig. 7.9): latency-oblivious and latency-bounded (Fig. 7.10). The latency-oblivious management continuously applies lower voltage and frequency to the least-used router for a fixed number of streams, without checking the run-time latency. This type of experiment is intended to demonstrate the energy reduction in cluster-level reconfiguration. The latency-bounded management, on the other hand, monitors the actual stream latency as in the CDDR algorithm (Section 5.3.3). In particular, a latency boundary (L_l; design-specific) is set as the delay of the largest communication flow CF_M among all nodes. The cell agents will trace and report the occurrences of the starting and ending timestamps of CF_M. The cluster agent compares the run-time latency L_r of CF_M with the latency boundary L_l. It will apply lower voltage and frequency to the least-used router until the boundary is reached or exceeded for the first time.

The per-stream energy consumption for the MPEG application with latency-oblivious simulation (for 45 streams) is depicted in Fig. 7.11. It can be observed that the per-stream energy is gradually decreasing as lower voltage and frequency are applied to individual routers. In particular, the 45th stream's energy consumption is 87.4% and 92.6% of the initial stream for energy-aware and random mapping respectively. In addition, the energy consumption of the energy-aware mapping is significantly lower than that of the random mapping (36.9% in terms of the initial stream, 34.8% in terms of the 45th stream; Fig. 7.11).

Similar trend is observed from the experiment with the BASIZ application. The per-stream energy consumption is illustrated in Fig. 7.12. The energy consumption of the 45th stream is 87.2% and 93.5% of the initial stream for the energy-aware and random mapping respectively. The energy consumption of the energy-aware mapping is 90.1% (for the initial stream) and 93.3% (for the 45th stream) of the corresponding stream in random mapping.

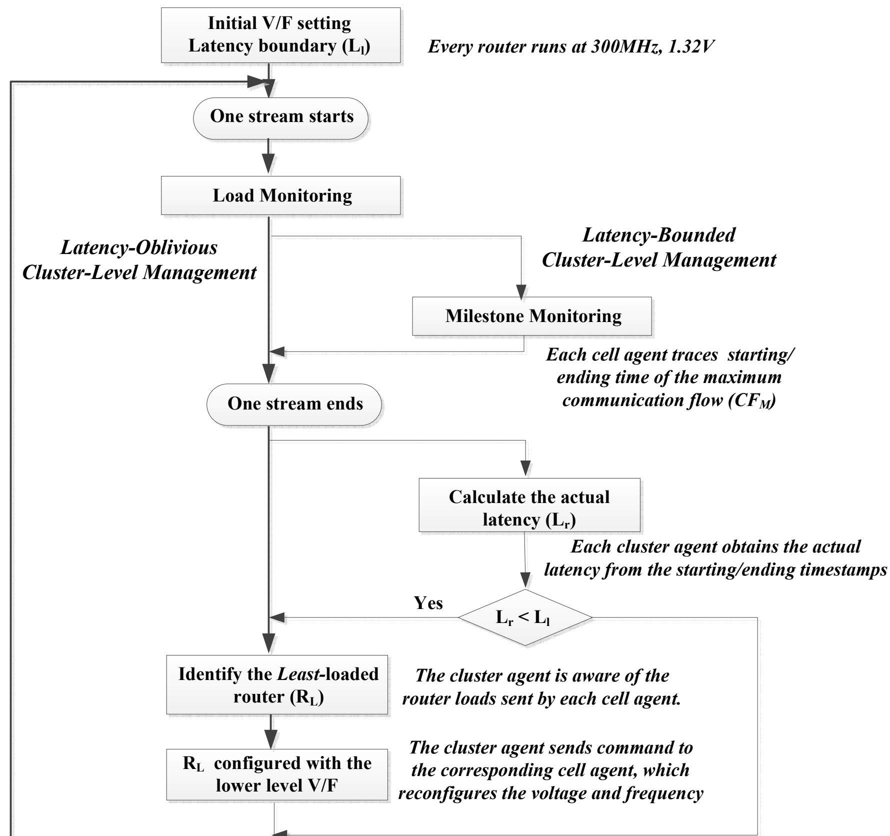


Figure 7.10: Minimizing Individual Router Energy with Latency-Oblivious and Latency-Bounded Cluster-Level Energy Management

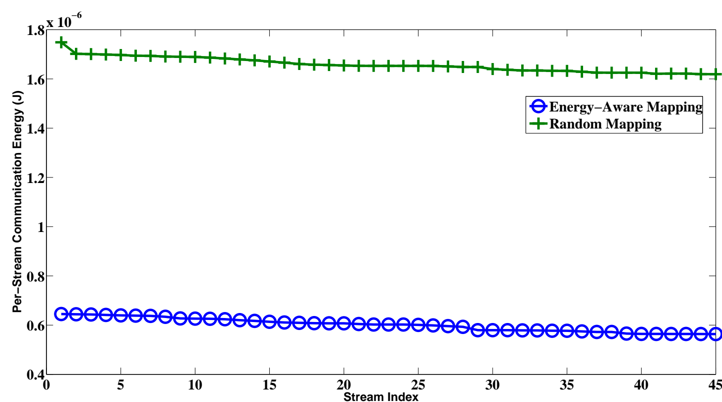


Figure 7.11: Per-Stream Energy Consumption in Latency-Oblivious Cluster-Level Energy Management (MPEG)

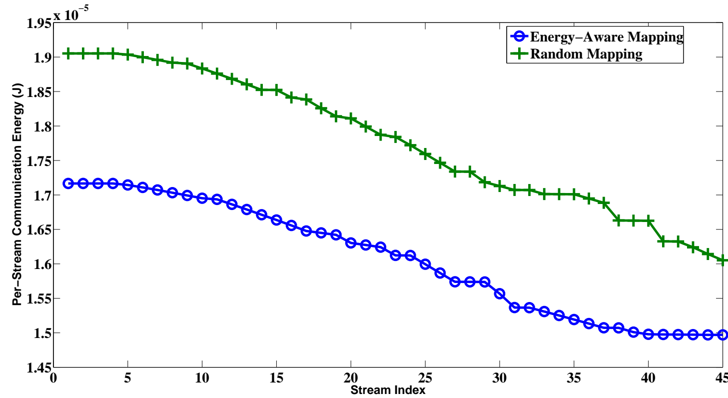


Figure 7.12: Per-Stream Energy Consumption in Latency-Oblivious Cluster-Level Energy Management (BASIZ)

In latency-bounded experiments, the performance awareness in hierarchical energy management can be analyzed and compared, as illustrated by Table 7.2. To highlight the comparison, the energy is normalized with the energy consumption of the first stream under energy-aware mapping. The latency is normalized with the latency boundary. The boundary is chosen as experimental values, as only the communication patterns of the two applications are available without the full processing details. From Table 7.2, it can be observed that the cluster-level energy management reaches the latency boundary with reduced energy consumption, for both applications under energy-aware mapping. The self-adaptation process takes a number of iterations (the exact number is application dependent). The stabilized energy consumption under the energy-aware mapping is around 87% of the initial stream for the two applications. With the random mapping, on the other hand, the latency is already higher than the boundary in the first stream (highest voltage and frequency setting), thus the self-adaptation process immediately stops.

The energy management experiments have confirmed the contribution of both the platform-level and cluster-level self-adaptation in energy saving. The total savings (stabilized energy in energy-aware mapping vs. initial energy with random mapping) are 67.8% for MPEG and 21.4% for the BASIZ. It should be noted that the relative saving of energy-aware mapping differs significantly upon the quality of random mapping. The cluster-level energy management, specifically the CDDR, provides effective energy saving while directly

Table 7.2: Per-Stream Energy and Latency in Latency-Bounded Cluster-Level Energy Management

Appli -cation	Latency Boundary	Mapping	Adaptation Ending Stream	Stabilized Energy (Normalized)	Stabilized Latency (Normalized)
MPEG	36000ns	Energy -aware	43rd	87.4%	96.3%
MPEG	36000ns	Random	1st	271.1%	479.3%
BASIZ	550000ns	Energy -aware	45th	87.2%	93.6%
BASIZ	550000ns	Random	1st	127.3%	118.7%

monitoring the performance (i.e., the latency boundary).

7.4.3 Fault-Tolerance Result

To analyze the dynamic clusterization, for each application, a number of failures are randomly injected into processors(Table 7.3). Random distribution of processor failures is usually assumed (e.g., [22]). Upon receiving the fault status report, the platform agent performs energy-aware remapping, a modified version of tree-based mapping algorithm (Section 6.4). In detail, each cluster's mesh area is transformed into an extended tree including the broken cells and all the spares. As illustrated in Fig. 7.13, the same mapping process as in tree-based mapping algorithm follows. The only modification is that, when the mapping proceeds to a broken cell, the next cell in the tree will be used instead. In this manner, spares will be mapped into each cluster with the total energy consumption minimized. The platform agent performs dynamic clusterization based on the remapping, with RLT, CLT, CIR and RT updated (Section 7.2). Afterwards, each cluster agent performs the same cluster-level energy management (both latency-oblivious and latency-bounded) as in Section 7.4.2.

The per-stream energy consumption under latency-oblivious cluster-level management is depicted in Fig. 7.14 for both applications. To highlight the relative energy saving in each application, the energy figure is normalized with the energy consumption of the initial stream in the corresponding application (fault-free case). It can be observed from Fig. 7.14 that the agent-based

Table 7.3: Injected Processor Failures and Utilized Spares (as intra-cluster locations)

Application	Failure Nodes	All Spares	Utilized Spares
MPEG 5*5 cluster	3,11, 12	0,1,2,4	0,1,2
BASIZ 6*5 cluster	6,14,15	0,12,18,24	12,18,24

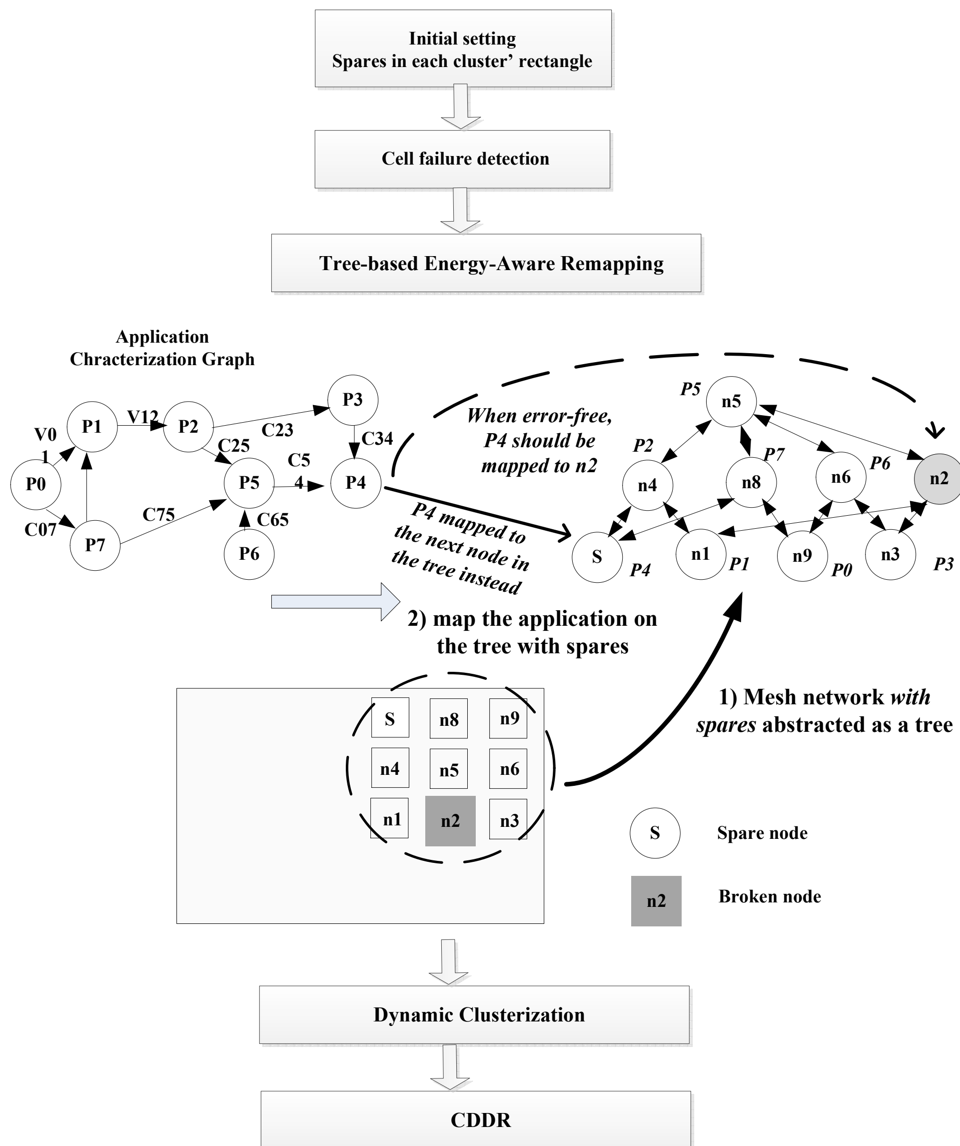


Figure 7.13: Energy-Aware Remapping for Dynamic Clusterization

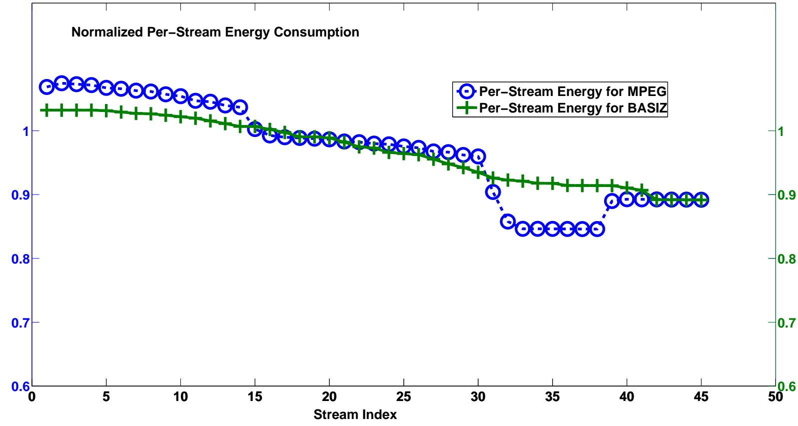


Figure 7.14: Per-stream Energy Consumption in Dynamically-Reconfigured Clusters (Latency-Oblivious Management)

Table 7.4: Comparison of Stabilized Energy and Latency in Fault-Injected and Fault-Free Clusters (Latency-Bounded Management)

Application	Fault Existence	Adaptation Ending Stream	Stabilized Energy	Stabilized Latency
MPEG	Fault-free	43rd	87.4%	96.3%
MPEG	3 processors	13rd	103.68%	95.8%
BASIZ	Fault-free	45rd	87.2%	93.6%
BASIZ	3 processors	28th	94.8%	98.9%

self-adaptation continues its proper functioning in face of processor failures, thanks to the dynamic clusterization. For latency-bounded simulation (Table 7.4), it can be further observed that the occurences of faults do not incur much energy overhead, due to the energy-aware remapping. With the same latency boundaries as in Table 7.2, the stabilized per-stream energy values of fault-injected clusters (normalized to the initial stream energy of the fault-free cases) are moderately increased from those in fault-free clusters (8.7% for the BASIZ application, 18.6% for the MPEG application).

7.5 Overheads

As presented in Section 4.3, H2A system architecture provides physical scalability via SW/HW co-design of agents. On the implemented SAA-NoC, the overheads of agents and supporting structures can be quantitatively analyzed. The platform agent is implemented as software, as a dedicated processor with the RLT (resource look-up table) mapped in the local memory. The cluster agent, as SW/HW co-design, includes both a thread running on a processor and a hardware-based CLT (cluster look-up table). The cell agent is hardware circuits wrapped in each NoC node. To examine whether the agent subsystem is physically scalable on the implemented SAA-NoC, different analysis for SW/HW structures needs to be followed. For software design, the major overhead is in terms of the timing penalty, or the relative amount of time during which a processor is occupied for agent services. For the platform agent, the energy-aware mapping will be analyzed in terms of its timing overhead on a $300MHz$ Leon 3 processor. For the fault-triggered dynamic clusterization, the major overhead on the platform level is also the energy-aware remapping as a large amount of data processing is required. In addition, as the platform agent requires a memory-mapped RLT, the relative memory usage is studied. For the cluster agent, as the CDDR algorithm is running as a thread on a processor, the software overhead is estimated by the timing overhead of the thread. The hardware overhead, i.e., the CLT, is estimated in terms of its silicon area with $65nm$ technology. Similarly, the overhead of each cell agent is measured by its area.

The results are summarized in in Table 7.5. Both the software and hardware overheads are minimal. The timing overhead of the energy-aware mapping for a 30-node cluster is only 33% of a MPEG stream [67]. The timing overhead is similar for the initial mapping and the remapping (with the same number of nodes), as the remapping follows the same tree-model-based searching algorithm with only additional checking on nodes' fault status (Fig. 7.13). Since the mapping is issued rarely (for the initial configuration or when permanent faults occur), the overhead has very small influence on the performance. The timing overhead of the CDDR algorith accounts even less in a MPEG stream's delay (0.17%). The hardware overheads, for either the CLT or the cell agent, are very small compared to the existing NoC area (only 2.2% and 0.7% of a 64-bit router). Importantly, all these overheads are scalable

with more number of cells in the NoC, due to the cluster-based management. Firstly, the energy-aware mapping mainly addresses the task mapping within each application (Section 5.2; assigning rectangles to each application has much lower complexity), whose overhead does not grow with the number of nodes in the NoC. The overhead of CDDR also depends on the cluster size, which is determined by the application size rather than the system size. For instance, the area overhead of CLT is proportional to the number of cells in each cluster, intead of cells in the whole system. The cell agent, which is a modularized design integrating all required local monitors, remains the same in system expansion.

7.6 Chapter Summary

As a proof-of-concept prototype, an implementation of SAA-NoC was elaborated in this chapter. Both adaptive energy management services and dynamic clusterization for fault tolerance were implemented upon a RTL NoC simulator with Leon 3 processors and shared memory. The platform, cluster and cell agents were synthesized with SW/HW co-design, demonstrating the hierarchical implementation guidelines of the H2A system architecture. Supporting structures for self-adaptation on each agent level, i.e., CLT, RLT and CIR, were also implemented. Experiments with one image processing and one video encoding application confirmed the energy saving of the hierarchical energy management. In particular, both platform-level energy-aware mapping and cluster-level CDDR significantly contribute to the energy minimization. In terms of fault-tolerance, the dynamic clusterization enables the reorganization of clusters using appropriate spares when certain processors fail. The energy-aware remapping minimizes the energy consumption for the dynamically reconfigured clusters. Last but not least, the overhead analysis of agents and the supporting structures showed minimal software overhead (in terms of timing) and hardware overhead (in terms of area), thus verified the physical scalability of the H2A design paradigm.

Table 7.5: Overheads of Agents and Supporting Structures

Type	Description	Value	Analysis	Scalability
SW	Timing overhead of Energy-aware Mapping on the platform agent	$\approx 11ms$ (task mapping of 30 processors /one cluster)	small 33% of a MPEG stream [67] (only required for initial configuration)	the overhead determined by application size, not system size, thus scalable
SW /memory	Relative RLT Overhead in the local memory	110 memory lines (one for each cell)	very small 0.7% of each 64KB local memory	linear to the system size
SW	Timing overhead of CDDR algorithm on the cluster agent	$\approx 56us$	very small 0.17% of a MPEG stream	the overhead determined by the cluster size, not the system size, thus scalable
HW	Area overhead of CLT 32 entries, each 16 bits	9451 um^2	minimal only 2.2% of a 64b router	not grow with the system size, thus scalable
HW	Cell agent	2972 um^2	only 0.7% of a 64b router	remain constant with system expansion

Chapter 8

Conclusion

Hierarchical Agent-based Adaptation (H2A) design paradigm is a systematic solution, which brings the self-adaptive features into parallel embedded systems. It major thrust is to improve the design productivity for scalable, energy efficient and dependable system development, in contrast to ad-hoc methods and techniques. The thesis successfully achieved the formulated objectives:

1. Identification and motivation of the design paradigm shift towards Self-Aware and Adaptive (SAA) systems.
2. Proposing a scalable design approach and system architecture for SAA systems.
3. Prototype and implementation of a SAA system.

8.1 Design Era of SAA systems

The thesis has established the SAA design as a dedicated design dimension, fulfilling the first objective. The paradigm shift is inspired by the proliferation of monitoring and reconfiguration techniques in parallel embedded systems, addressing performance, energy efficiency and dependability. Exploiting the system-level design principle of orthogonalization, a separate design layer for monitoring and reconfiguration services has been proposed, to address the design productivity via reuse and portability. The SAA concept has been compared with autonomic computing and reconfigurable computing. While autonomic computing spreads over a broad spectrum of behaviors and objectives, SAA computing focuses on the architectural integration of self-awareness and

adaptation. Reconfigurable computing enables the adaptation process, but does not directly address the distinct phases of self-monitoring and adaptive reconfiguration.

The thesis has studied the three key processes in the closed-loop feedback universal in a SAA system: monitoring, decision-making and reconfiguration. With energy efficiency as the primary objective, the thesis presented the design and comparison of centralized, clustered and distributed run-time management architectures. Distributed self-adaptive management is more effective in local optimization, while incurring physical design complexity. The clustered management provides a tradeoff between local optimization, global awareness and design complexity. The analysis has identified the need of a systematic integration of self-adaptive techniques on different architectural levels.

8.2 Hierarchical Agent Architecture

A system architecture to design SAA systems, hierarchical agent-based adaptation (H2A), was presented in the thesis. From the design paradigm's perspective, the self-adaptation architecture with platform, cluster and cell agents can be applied on different types of systems, thus enabling a portable approach. As a system architecture, the functional partitioning among high-level and low-level agents minimizes the burden on the centralized decision-maker while still supporting fine-grained local status monitoring. In terms of overheads, the low-level agents are implemented as hardware and wrapped within each modularized local component. Thus the relative overhead remains constant when the system expands. High-level agents are implemented as software to perform diverse operations without adding hardware overheads for each operation. With design reuse, functional partitioning and SW/HW co-design, H2A achieves scalability for the agent subsystem in terms of performance, overhead and design effort.

The thesis has presented hierarchical energy management and dynamic clusterization as two major services with the H2A architecture. For energy management, the platform agent is designed as a software controller, performing energy-aware application mapping. Each cluster agent, with SW/HW co-design, enables intra-cluster energy management with CDDR (clustered decision-making with distributed reconfiguration). CDDR exploits coarse-grained performance awareness of the clustered decision-making, while being

able to fine-tune the status of local resources (e.g., routers and channels) with distributed reconfiguration. Each cell agent, as a modularized hardware controller, provides hybrid monitoring (both local buffer loads and application milestones). The joint effort of all levels of agents significantly reduces the energy consumption of the system, while meeting the performance constraints of the applications. As modern parallel systems are highly susceptible to unpredictable errors and failures, dynamic clusterization against processor failures has been proposed for dependable computing. In particular, H2A enables any cell to be allocated to any cluster at the run-time, eliminating the influence of processor failures on the statically assigned clusters. To enable dynamic clusterization, three-level supporting structures have been designed, in addition to a full-mesh-based physically separate monitoring network.

8.3 Self-Aware and Adaptive NoC Implementation

The thesis has presented a RTL implementation of a Self-Aware and Adaptive NoC (SAA-NoC). The implementation integrates both hierarchical energy management and dynamic clusterization against processor failures. The platform agent is implemented in software running on a dedicated processor. The RLT is implemented in the local memory of the platform agent. Each cluster agent is implemented as a thread running on a processor. The CLT is implemented as a small and reconfigurable hardware unit affiliated to the cluster agent. Each cell agent is a hardware unit wrapped within a NoC node, with reconfigurable CIR, RT and MIR (milestone instruction register). With multi-Vdd 65nm CMOS libraries, hierarchical energy management achieves 67.8% and 21.4% energy saving for an image processing and a video encoding application respectively. In case of random processor failures, the system correctly updates the clusterization with suitable replacement. The energy consumption is increased by only 8.7% and 18.6% for the two applications. In terms of SW overhead, the energy-aware mapping function on the platform agent accounts for 33% of a MPEG stream's delay, which is rarely incurred as the mapping is only needed for the initial configuration or when permanent processor failures have occurred. In addition, the cluster-level energy management thread only accounts for 0.17% of a MPEG stream's delay. In terms of HW overhead, a CLT is 2.2% of a 64-bit NoC router, while each cell agent only accounts for 0.7%.

8.4 From Parallel to Distributed Computing: Future Work

As the foundational work in the development of H2A-based SAA systems, the thesis's research has fostered a wide range of studies covering the self-awareness and adaptivity concepts, system architectures and formal methods. Though not having reached a mature stage of research outputs, some of our related works reported the latest progress on formal specification [41, 40], fault tolerance of link-level errors [3] and the application of H2A on distributed systems [35]. The on-going research focuses on the following three directions:

1. Improving the system dependability by tolerating agent faults. The paradigm overview in Chapter 4 has proposed that higher level agents should monitor the error status of the affiliated lower level agents, but such techniques are not elaborated. Thus one on-going research is to design a fault-tolerant inter-agent communication protocol, which can detect the run-time faults on each level of agents. A possible protocol is to use regular testing messages from one higher-level agent to all its affiliated lower-level agents. A healthy agent should be able to respond with the correct reply within an expected time period.

2. Formal validation of agent and resource status. On a hierarchical agent-based system, run-time reconfiguration is applied simultaneously to distributed cells, clusters and the platform, whose states in turn influence the decision-making of agents on each level. A systematic method for specifying and validating such a system is highly beneficial to the development of the H2A paradigm. A high-level language to specify the agent behaviors and state transition of resources has been presented in [41, 40]. The future work aims to formulate a complete approach for formally validating the state transition of agents and resources.

3. Applying H2A on Distributed Embedded Systems (DES). A DES has higher diversity in communication channels and end devices than multiprocessor systems. On a heterogeneous DES, ad-hoc monitoring and reconfiguration techniques will lead to significant design complexity, thus the H2A design paradigm should be utilized for its scalability and portability. [42] has presented an initial study on the smart house architecture with hierarchical agents. The platform, cluster and cell agents are

assigned on the application, service and sensor layers respectively. Services to enhance the dependability of the monitoring and surveillance in the smart house are exemplified. The research aims to design hierarchical agent-based adaptation as a generic architecture for pervasively connected DES, e.g., CPS (Cyber-Physical Systems).

The thesis has innovated a scalable and portable system architecture, H2A, for the design of self-awareness and self-adaptivity in embedded systems. It has demonstrated the feasibility, effectiveness and efficiency of H2A by extensive architectural-level design and experiments. The analysis on hierarchical energy management and dynamic clusterization services has shown the energy-efficiency and dependability of applying a systematic approach on the state-of-the-art parallel embedded systems. The paradigm's future promise can be found in a broader scope of parallel and distributed systems.

Bibliography

[1] Noxim - the noc simulator. http://sourceforge.net/projects/noxim/.

[2] Mohammad Abdullah Al Faruque, Rudolf Krist, and Jörg Henkel. Adam: run-time agent-based distributed application mapping for on-chip communication. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 760–765, New York, NY, USA, 2008. ACM.

[3] Mohammad Asad, Liang Guang, Ahmed Hemani, Kolin Paul, Juha Plosila, and H Tenhunen. Energy-aware fault-tolerant network-on-chips for addressing multiple traffic classes. In *15th Euromicro Conference on Digital System Design*, page to appear, 2012.

[4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, U.C.Berkeley, 2006.

[5] Maryam Ashouei. *Algorithms and Methodology for Post-Manufacture Adaptation to Process Variations and Induced Noise in Deeply Scaled CMOS Technologies*. PhD thesis, Georgia Institute of Technology, 2007.

[6] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[7] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Des. Test*, 17(1):68–83, 2000.

[8] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Digest of Technical Papers. IEEE ISSCC 2008*, pages 88–89,598, 2008.

[9] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proc. Design Automation Conference*, pages 338–342, 2003.

[10] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proc. Design Automation Conference*, pages 746–749, 2007.

[11] Shekhar Borkar, Norman P. Jouppi, and Per Stenstrom. Microprocessors in the era of terascale integration. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 237–242, San Jose, CA, USA, 2007. EDA Consortium.

[12] K. Bousias, L. Guang, C. R. Jesshope, and M. Lankamp. Implementation and evaluation of a microthread architecture. *J. Syst. Archit.*, 55(3):149–161, March 2009.

[13] J.-M. Chabloz and A. Hemani. A flexible communication scheme for rationally-related clock frequencies. In *Proc. IEEE Int. Conf. Computer Design ICCD 2009*, pages 109–116, 2009.

[14] J. M. Chabloz and A. Hemani. Distributed DVFS using rationally-related frequencies and discrete voltage levels. In *Proc. ACM/IEEE Int Low-Power Electronics and Design (ISLPED) Symp*, pages 247–252, 2010.

[15] Jean-Michel Chabloz and Ahmed Hemani. *Power Management Architecture in McNoC*, pages 55–80. Springer New York, 2012.

[16] V. Chandra and R. Aitken. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale cmos. In *Proc. IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems DFTVS '08*, pages 114–122, 2008.

[17] Xiaowen Chen, Zhonghai Lu, A. Jantsch, and Shuming Chen. Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. In *Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, pages 39–44, 2010.

[18] Xiaowen Chen and Lu Zhonghai. *DME User Guide*. KTH, School of Information and Communication Technology, release 4.0 edition, Feb 2010.

[19] W.H. Cheng and B.M. Baas. Dynamic voltage and frequency scaling circuits with two supply voltages. In *Proc. ISCAS 2008*, pages 1236–1239, 2008.

[20] Chen-Ling Chou and R. Marculescu. Contention-aware application mapping for network-on-chip communication architectures. In *Proc. IEEE International Conference on Computer Design ICCD 2008*, pages 164–169, October 12–15, 2008.

[21] Calin Ciordas, Andreas Hansson, Kees Goossens, and Twan Basten. A monitoring-aware network-on-chip design flow. *Journal of Systems Architecture - Embedded Systems Design*, 54(3-4):397–410, 2008.

[22] J. H. Collet, M. Psarakis, P. Zajac, D. Gizopoulos, and A. Napieralski. Comparison of fault-tolerance techniques for massively defective fine- and coarse-grained nanochips. In *Proc. MIXDES-16th Int. Conf. Mixed Design of Integrated Circuits & Systems MIXDES '09*, pages 23–30, 2009.

[23] Nikolaus Correll, Nikos Arechiga, Adrienne Bolger, Mario Bollini, Ben Charrow, Adam Clayton, Felipe Dominguez, Kenneth Donahue, Samuel Dyar, Luke Johnson, Huan Liu, Alexander Patrikalakis, Timothy Robertson, Jeremy Smith, Daniel Soltero, Melissa Tanner, Lauren White, and Daniela Rus. Building a distributed robot garden. In *IROS'09: Proceedings of the 2009 IEEE/RSJ international conference on Intelligent robots and systems*, pages 1509–1516, Piscataway, NJ, USA, 2009. IEEE Press.

[24] R. Dafali and J.-P. Diguet. Self-adaptive network interface (SANI): Local component of a noc configuration manager. In *Proc. Int. Conf. Reconfigurable Computing and FPGAs ReConFig '09*, pages 296–301, 2009.

[25] William J. Dally. Computer architecture is all about interconnect. HPCA panel, Feb. 2002.

[26] William James Dally and Brian Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2004.

[27] S. Das, C. Tokunaga, S. Pant, W.-H. Ma, S. Kalaiselvan, K. Lai, D.M. Bull, and D.T. Blaauw. RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance. *IEEE JSSC*, 44(1):32–48, 2009.

[28] Robert Dick. Embedded system synthesis benchmarks suite (e3s). http://ziyang.eecs.umich.edu/ dickrp/e3s/.

[29] L. Fiorin, G. Palermo, and C. Silvano. MPSoCs run-time monitoring through Networks-on-Chip. In *Proc. DATE '09. Design, Automation & Test in Europe Conference & Exhibition*, pages 558–561, 2009.

[30] The International Technology Roadmap for Semiconductors. Itrs 2010 update. Technical report, 2010.

[31] The International Technology Roadmap for Semiconductors. ITRS executive summary 2011 edition. Technical report, 2011.

[32] Jiri Gaisler, Edvin Catovic, Marko Isomaki, Kristoffer Glembo, and Sandi Habinc. *GRLIB IP Core User's Manual*. Gaisler Research, version 1.0.20 edition, Feb 2009.

[33] José C. García, Juan A. Montiel-Nelson, and Saeid Nooshabadi. High performance cmos dual supply level shifter for a 0.5v input and 1v output in standard 1.2v 65nm technology process. In *ISCIT'09*, pages 963–966. IEEE Press, 2009.

[34] J. Gjanci and M.H. Chowdhury. A hybrid scheme for on-chip voltage regulation in system-on-a-chip (SOC). *IEEE Transactions on VLSI*, 19(11):1949 –1959, nov. 2011.

[35] Liang Guang, Rajeev Kanth, Juha Plosila, and Hannu Tenhunen. Hierarchical monitoring in smart house: Design scalability, dependability and energy-efficiency. *Information Science and Management Engineering (CISME)*, to appear.

[36] Liang Guang, Ethiopia Nigussie, Jouni Isoaho, Pekka Rantala, and Hannu Tenhunen. Interconnection alternatives for hierarchical monitoring communication in parallel socs. *Microprocess. Microsyst.*, 34(5):118–128, August 2010.

[37] Liang Guang, Ethiopia Nigussie, Lauri Koskinen, and Hannu Tenhunen. Autonomous dvfs on supply islands for energy-constrained noc communication. In *Proceedings of ARCS 09*, LNCS 5545, March 2009.

[38] Liang Guang, Ethiopia Nigussie, Juha Plosila, Jouni Isoaho, and Hannu Tenhunen. HLS-DoNoC: High-level simulator for dynamically organizational NoCs. In *DDECS*, pages 89–94, 2012.

[39] Liang Guang, Ethiopia Nigussie, Juha Plosila, Jouni Isoaho, and Hannu Tenhunen. Survey of self-adaptive nocs with energy-efficiency and dependability. *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 3(2):1–22, 2012.

[40] Liang Guang, Juha Plosila, Jouni Isoaho, and Hannu Tenhunen. Hierarchical agent monitored parallel on-chip system: A novel design paradigm and its formal specification. *International Journal of Embedded and Real-Time communication Systems (IJERTCS)*, 1(2):86–105, 2010.

[41] Liang Guang, Juha Plosila, Jouni Isoaho, and Hannu Tenhunen. *HAM-SoC: A Monitoring-Centric Design Approach for Adaptive Parallel Computing*, pages 135 – 164. Taylor & Francis/CRC Press, 2011.

[42] Liang Guang, Bo Yang, Juha Plosila, Jouni Isoaho, and Hannu Tenhunen. Hierarchical agent monitoring design platform - towards self-aware and adaptive embedded systems. In *PECCS*, pages 573–581, 2011.

[43] Nadim El Guindi and Pascal Elsener. Network on chip: PANACEA - A Nostrum integration. Technical report, Swiss Federal Institute of Technology Zurich, February 2005.

[44] M.S. Gupta, J.L. Oatley, R. Joseph, Gu-Yeon Wei, and D.M. Brooks. Understanding voltage variations in chip multiprocessors using a distributed power-delivery network. In *Proc. Design, Automation & Test in Europe Conference & Exhibition DATE '07*, pages 1–6, 2007.

[45] P. Hazucha, G. Schrom, Jaehong Hahn, B.A. Bloechel, P. Hack, G.E. Dermer, S. Narendra, D. Gardner, T. Karnik, V. De, and S. Borkar. A 233-mhz 80%-87% efficient four-phase dc-dc converter utilizing air-core inductors on package. *IEEE Journal of Solid-State Circuits*, 40(4):838–845, 2005.

[46] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats for software performance and health. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 347–348, New York, NY, USA, 2010. ACM.

[47] Paul Horn. Autonomic computing: IBM's perspective on the state of information technology, 2001.

[48] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. Van Der Wijngaart. A 48-Core IA-32 Processor in 45 nm CMOS using On-Die Message-Passing and DVFS for Performance and Power Scaling. *IEEE Journal of Solid-State Circuits*, 46(1):173–183, 2011.

[49] Jingcao Hu and R. Marculescu. Energy and performance-aware mapping for regular noc architectures. *IEEE Transactions on CAD*, 24(4):551–562, 2005.

[50] Pablo Iñigo Blasco, Fernando Diaz-del Rio, Ma Carmen Romero-Ternero, Daniel Cagigas-Muñiz, and Saturnino Vicente-Diaz. Robotics software frameworks for multi-agent robotic systems development. *Robot. Auton. Syst.*, 60(6):803–821, June 2012.

[51] Jouni Isoaho, Seppo Virtanen, and Juha Plosila. Current challenges in embedded communication systems. *IJERTCS*, 1(1):1–21, 2010.

[52] Axel Jantsch. Nostrum home. http://www.ict.kth.se/nostrum/.

[53] Axel Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation.* Morgan Kaufmann Publishers Inc., 2003.

[54] Axel Jantsch and Hannu Tenhunen. *Networks on Chip.* Kluwer Academic Publishers, 2003.

[55] A.B. Kahng, Bin Li, Li-Shiuan Peh, and K. Samadi. Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration. In *Proc. DATE '09*, pages 423–428, 2009.

[56] N. Karimi, A. Alaghi, M. Sedghi, and Z. Navabi. Online network-on-chip switch fault detection and diagnosis using functional switch faults. *Journal of Universal Computer Science*, 14(22):3716–3736, 2008.

[57] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[58] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on CAD*, 19(12):1523–1543, Dec. 2000.

[59] Gul Khan and Usman Ahmed. Cad tool for hardware software co-synthesis of heterogeneous multiple processor embedded architectures. *Design Automation for Embedded Systems*, 12:313–343, 2008.

[60] S. Khursheed, Shida Zhong, R. Aitken, B. M. Al-Hashimi, and S. Kundu. Modeling the impact of process variation on resistive bridge defects. In *Proc. IEEE Int. Test Conf. (ITC)*, pages 1–10, 2010.

[61] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.

[62] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proc. HPCA 2008*, pages 123–134, 16–20 Feb. 2008.

[63] Avinash Karanth Kodi, Ashwini Sarathy, and Ahmed Louri. Adaptive channel buffers in on-chip interconnection networks- a power and performance analysis. *IEEE Transactions on Computers*, 57(9):1169–1181, 2008.

[64] Avinash Karanth Kodi, Ashwini Sarathy, Ahmed Louri, and Janet Wang. Adaptive inter-router links for low-power, area-efficient and reliable

Network-on-Chip (NoC) architectures. In *Proc. of ASP-DAC '09*, pages 1–6, Piscataway, NJ, USA, 2009. IEEE Press.

[65] S. Kumar, A. Jantsch, J. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrja, and A Hemani. A network on chip architecture and design methodology. In *Proc. of ISVLSI '02*, pages 105–112, 2002.

[66] Vipin Kumar. *Introduction to Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[67] Didier Le Gall. MPEG: a video compression standard for multimedia applications. *Commun. ACM*, 34(4):46–58, April 1991.

[68] Teijo Lehtonen, Pasi Liljeberg, and Juha Plosila. Online reconfigurable self-timed links for fault tolerant noc. *VLSI Design*, 2007:13, 2007.

[69] Bin Li, Li-Shiuan Peh, and P. Patra. Impact of process and temperature variations on network-on-chip design exploration. In *Proc. of NoCS 2008*, pages 117–126, 2008.

[70] Sheng Li, Jung Ho Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. MICRO-42*, pages 469–480, 2009.

[71] G. Liang and A. Jantsch. Adaptive power management for the on-chip communication network. In *Proc. 9th EUROMICRO Conf. Digital System Design: Architectures, Methods and Tools DSD 2006*, pages 649–656, 2006.

[72] Wei-Ming Lin, Chao-Chyun Chen, and Shen-Iuan Liu. An all-digital clock generator for dynamic frequency scaling. In *Proc. of International Symposium on VLSI Design, Automation and Test*, pages 251–254, 2009.

[73] Z. Lu and A. Jantsch. Tdm virtual-circuit configuration for network-on-chip. *IEEE Transactions on VLSI Systems*, 16(8):1021–1034, 2008.

[74] Z. Lu, A. Jantsch, E. Salminen, and C. Grecu. Network-on-chip benchmarking specification part 2: Microbenchmark specification version 1.0. Technical report, OCP International Partnership Association, Inc., May 2008.

[75] Zhonghai Lu, Rikard Thid, Mikael Millberg, and Axel Jantsch. NNSE: Nostrum network-on-chip simulation environment. In *In Proc. of SSoCC*, 2005.

[76] S. Madduri, R. Vadlamani, W. Burleson, and R. Tessier. A monitor interconnect and support subsystem for multicore processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 761–766, 2009.

[77] C. Marcon, A. Borin, A. Susin, L. Carro, and F. Wagner. Time and energy efficient mapping of embedded applications onto nocs. In *Proc. Asia and South Pacific Design Automation Conference the ASP-DAC 2005*, volume 1, pages 33–38, January 18–21, 2005.

[78] César Augusto Missio Marcon, Edson Ifarraguirre Moreno, Ney Laert Vilar Calazans, and Fernando Gehm Moraes. Comparison of network-on-chip mapping algorithms targeting low energy consumption. *IET Computers & Digital Techniques*, 2(6):471–482, 2008.

[79] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33:92–99, November 2005.

[80] H. Matsutani, M. Koibuchi, Daihan Wang, and H. Amano. Run-time power gating of on-chip routers using look-ahead routing. In *Proc. of ASP-DAC 2008*, pages 55–60, 2008.

[81] I. Miro Panades and A. Greiner. Bi-synchronous fifo for synchronous circuit communication well suited for network-on-chip in gals architectures. In *Proc. of NOCS 2007*, pages 83–94, 2007.

[82] Hamid Reza Naji and B. Earl Wells. On incorporating multi agents in combined hardware/software based reconfigurable systems– a general architectural framework. In *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory*, pages 344– 348, 2002.

[83] Hamid Reza Naji, B. Earl Wells, and Letha Etzkorn. Creating an adaptive embedded system by applying multi-agent techniques to reconfig-

urable hardware. *Future Gener. Comput. Syst.*, 20(6):1055–1081, August 2004.

[84] C.A. Nicopoulos, Dongkook Park, Jongman Kim, N. Vijaykrishnan, M.S. Yousif, and C.R. Das. Vichar: A dynamic virtual channel regulator for network-on-chip routers. In *Proc. MICRO-39 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 333–346, 2006.

[85] E. Nigussie, S. Tuuna, J. Plosila, and J. Isoaho. Analysis of crosstalk and process variations effects on on-chip interconnects. In *Proc. Int System-on-Chip Symp*, pages 1–4, 2006.

[86] Ethiopia Nigussie, Juha Plosila, and Jouni Isoaho. Process variation tolerant on-chip communication using receiver and driver reconfiguration. In *ISQED*, pages 453–460, 2010.

[87] Erland Nilsson. Design and implementation of a hot-potato switch in a network on chip. Master's thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, IMIT/LECS 2002-11, Stockholm, Sweden, June 2002.

[88] Tammy Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Newnes, 2005.

[89] K. Nose, A. Shibayama, H. Kodama, M. Mizuno, M. Edahiro, and N. Nishi. Deterministic inter-core synchronization with periodically all-in-phase clocking for low-power multi-core socs. In *Proc. of ISSCC. 2005*, pages 296–599, 2005.

[90] U.Y. Ogras, R. Marculescu, and D. Marculescu. Variation-adaptive feedback control for networks-on-chip with multiple clock domains. In *Proc. of DAC 2008*, pages 614–619, 2008.

[91] Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.*, 51(2):180–199, February 2002.

[92] Dinesh Pamunuwa, Johnny Oberg, Li-Rong Zheng, Mikael Millberg, Axel Jantsch, and Hannu Tenhunen. A study on the implementation

of 2-d mesh based networks on chip in the nanoregime. *Integration - The VLSI Journal*, 38(1):3–17, 2004.

[93] Jan M. Rabaey. *System-on-chip challenges in the deep-sub-micron era*, chapter 1, pages 3–24. Kluwer Academic Publishers, 2004.

[94] Jan M. Rabaey. *Interconnect-Centric Design for Advanced SoC and NoC*, chapter System-on-Chip-Challenges in the Deep-Sub-Micron Era , A case for the network-on-a-Chip, pages 3–24. Springer US, 2005.

[95] Jan M. Rabaey. Scaling the power wall: Revisiting the low-power design rules. Keynote speech at SoC 07 Symposium, Nov. 2007.

[96] C. Roig, A. Ripoll, and F. Guirado. A new task graph model for mapping message passing applications. *IEEE Transactions on Parallel and Distributed Systems*, 18(12):1740–1753, 2007.

[97] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd edition)*. NJ: Prentice Hall, 2003.

[98] M. Saen, K. Osada, S. Misaka, T. Yamada, Y. Tsujimoto, Y. Kondoh, T. Kamei, Y. Yoshida, E. Nagahama, Y. Nitta, T. Ito, T. Kameyama, and N. Irie. Embedded SoC resource manager to control temperature and data bandwidth. In *Proc. Digest of Technical Papers. IEEE Int. Solid-State Circuits Conf. ISSCC 2007*, pages 296–604, 2007.

[99] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.

[100] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, 18(6):23–33, Nov.–Dec. 2001.

[101] Alberto Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.

[102] T. Santa, M. Auer, C. Sandner, and C. Lindholm. Switched capacitor dc-dc converter in 65nm cmos technology with a peak efficiency of 97 In *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*, pages 1351 –1354, may 2011.

[103] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Varius: A model of process variation and resulting timing errors for microarchitects. *IEEE Transactions on Semiconductor Manufacturing*, 21(1):3–13, 2008.

[104] T. Schattkowsky and W. Muller. Model-based design of embedded systems. In *Proc. Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 113–128, 2004.

[105] S. Shamshiri, P. Lisherness, Sung-Jui Pan, and Kwang-Ting Cheng. A cost analysis framework for multi-core systems with spares. In *Proc. IEEE International Test Conference ITC 2008*, pages 1–8, 2008.

[106] Li Shang, L. Peh, A. Kumar, and N.K. Jha. Thermal modeling, characterization and management of on-chip networks. In *Proc. 37th International Symposium on Microarchitecture MICRO-37 2004*, pages 67–78, 2004.

[107] Li Shang, Li-Shiuan Peh, and N.K. Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *Proc. of HPCA 2003*, pages 91–102, 2003.

[108] C. Silvano, W. Fornaciari, S. Crespi Reghizzi, G. Agosta, G. Palermo, V. Zaccaria, P. Bellasi, F. Castro, S. Corbetta, E. Speziale, D. Melpignano, J. M. Zins, D. Siorpaes, H. Hubert, B. Stabernack, J. Brandenburg, M. Palkovic, P. Raghavan, C. Ykman-Couvreur, A. Bartzas, D. Soudris, T. Kempf, G. Ascheid, H. Meyr, J. Ansari, P. Mahonen, and B. Vanthournout. Parallel paradigms and run-time management techniques for many-core architectures: The 2parma approach. In *Proc. 9th IEEE Int Industrial Informatics (INDIN) Conf*, pages 835–840, 2011.

[109] F. Sironi, A. Cuoccio, H. Hoffmann, M. Maggio, and M.D. Santambrogio. Evolvable systems on reconfigurable architecture via self-aware adaptive applications. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 176 –183, june 2011.

[110] V. Soteriou and Li-Shiuan Peh. Exploring the design space of self-regulating power-aware on/off interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 18(3):393–408, 2007.

[111] B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[112] Anthony John Stratakos. *High-efficiency low-voltage DC-DC conversion for portable applications.* PhD thesis, University of California, Berkeley, 1998.

[113] D. Sylvester, D. Blaauw, and E. Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *IEEE Design & Test of Computers*, 23(6):484–490, 2006.

[114] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[115] D.N. Truong, W.H. Cheng, T. Mohsenin, Zhiyi Yu, A.T. Jacobson, G. Landge, M.J. Meeuwsen, C. Watnik, A.T. Tran, Zhibin Xiao, E.W. Work, J.W. Webb, P.V. Mejia, and B.M. Baas. A 167-processor computational platform in 65 nm cmos. *IEEE Journal of Solid State Circuits*, 44(4):1130–1144, 2009.

[116] D. Truscan, J. Lilius, T. Seceleanu, and H. Tenhunen. A model-based design process for the segbus distributed architecture. In *Proc. Int. Conf. on Engineering of Computer-Based Systems*, pages 307–316, 2008.

[117] O.S. Unsal, J.W. Tschanz, K. Bowman, V. De, X. Vera, A. Gonzalez, and O. Ergin. Impact of parameter variations on circuits and microarchitecture. *IEEE Micro*, 26(6):30–39, 2006.

[118] C.H. van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1260–1265, 2009.

[119] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *IEEE JSSC*, 43(1):29–41, 2008.

[120] G. V. Varatkar, S. Narayanan, N. R. Shanbhag, and D. Jones. Sensor network-on-chip. In *Proc. Int System-on-Chip Symp*, pages 1–4, 2007.

[121] K. G. Verma, B. K. Kaushik, and R. Singh. Effects of process variation in VLSI interconnects - a technical review. *Microelectronics International*, 26(3):49–55.

[122] Jeffrey S. Vetter and Frank Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput.*, 63(9):853–865, 2003.

[123] Mengzhi Wang, T. Madhyastha, Ngai Hang Chan, S. Papadimitriou, and C. Faloutsos. Data mining meets performance evaluation: fast algorithms for modeling bursty traffic. In *Proc. 18th International Conference on Data Engineering*, pages 507–516, 2002.

[124] D. Wentzlaff, P. Griffin, H. Hoffmann, Liewei Bao, B. Edwards, C. Ramey, M. Mattina, Chyi-Chang Miao, J.F. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE MICRO*, 27(5):15–31, 2007.

[125] Paul E. West, Yuval Peress, Gary S. Tyson, and Sally A. McKee. Core monitors: monitoring performance in multicore processors. In *Proceedings of the 6th ACM conference on Computing frontiers*, CF '09, pages 31–40, New York, NY, USA, 2009. ACM.

[126] Frédéric Worm, Paolo Ienne, Patrick Thiran, and Giovanni De Micheli. A robust self-calibrating transmission scheme for on-chip networks. *IEEE Trans. Very Large Scale Integr. Syst.*, 13(1):126–139, 2005.

[127] Bo Yang, Liang Guang, Xu Thomas Canhao, Alexander Wei Yin, Tero Säntti, and Juha Plosila. Multi-application multi-step mapping method for many-core network-on-chips. In *Proc. of Norchip 2010*, pages 1–6, 2010.

[128] Bo Yang, Thomas Canhao Xu, Tero Säntti, and Juha Plosila. Tree-model based mapping for energy-efficient and low-latency network-on-chip. In *Proc. of 13th IEEE DDECS*, pages 189–192, 2010.

[129] A. Yin, L. Guang, P. Liljeberg, P. Rantala, J. Isoaho, and H. Tenhunen. Hierarchical agent based noc with dvfs techniques. *International Journal of Design, Analysis and Tools for Circuits and Systems*, 1(1):32–40, 2011.

[130] Lei Zhang, Yinhe Han, Qiang Xu, and Xiaowei Li. Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology. In *Proc. Design, Automation and Test in Europe DATE '08*, pages 891–896, 2008.

Appendix A

HLS-DoNoC: High-Level Simulator for Dynamically Organizational NoCs

Here we describe the system-level simulator (published in [38]) used for the early-stage design exploration in Chapter 3, 5 and 6. Written mostly in C++, it is able to simulate various dynamic cluster-based monitoring and reconfiguration techniques. Nodes in the network can be dynamically grouped into different clusters. Monitoring of various parameters and reconfiguration of voltage and frequency values can be applied to each cluster. Network status, communication performance and power/energy consumption are provided to facilitate the network design and analysis.

There exist many NoC simulators with different features and functions. In terms of system scope, McPAT [70] and GEMS [79] are both full system simulators. McPAT [70] provides detailed power, area and timing models for different components in a many-core system. GEMS [79] integrates comprehensive processor model and memory system simulator. These simulators usually require lengthy simulation to obtain accurate details for each system component. Our simulator (HLS-DoNoC) directly addresses dynamic organization and related monitoring/reconfiguration techniques in NoCs, thus simplifies the reprogramming and simulation process. The goal of HLS-DoNoC is to provide initial but fast analysis for various cluster-based dynamic management schemes, in order to narrow down the design space for follow-up circuit-level design and implementation. Compared to NoC-dedicated simulators (e.g. NNSE [75]

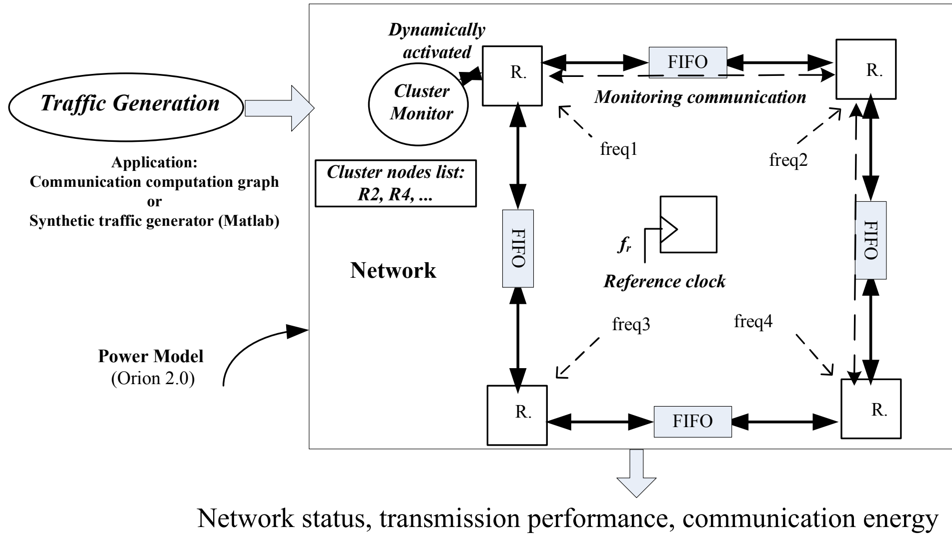


Figure A.1: Module Overview of Simulation Framework

and Noxim [1]), HLS-NoC extends their functions by supporting the adaptive monitoring/reconfiguration functions. Orion 2.0 [55] is a widely used tool for power and area modeling in NoCs. But it is not a network function simulator (i.e. it does not model the transmission process). HLS-DoNoC integrates Orion to estimate energy and power consumption (Section A.5).

A.1 Modular Overview

The simulator is composed of a network kernel, structures for dynamic clusterization and run-time reconfiguration, and traffic generators (Fig. A.1). The network kernel is composed of wormhole-based NoC routers. Any router can be attached with a monitor to supervisor a cluster, activating the affiliated cluster look-up table. Routers can run on different frequencies ratiochronous to a reference clock (Section A.4). The traffic generation supports both practical applications and synthetic traffic traces. Monitoring communication is simulated separately from the data communication. The simulator can give results in terms of network status (e.g. workload, congestion), performance (e.g. latency and throughput), as well as energy and power of any communication flow.

A.2 Network Kernel

In the network kernel, each router is modeled with wormhole-switching, and is composed of input buffer, routing and arbitration logic, and crossbar. The simulator can generate networks of any topologies. The size of input buffers can be configured. The router includes four pipeline stages: input, routing, crossbar traversal and FIFO writing (related to the synchronization between clusters; details in Section A.3). By default, the simulator uses X-Y routing.

The communication channels support both direct communication (when two ends are running on the same frequency and phase) and synchronizer-inserted communication (when two ends are allowed different frequencies and phases). The synchronizer is modeled as the FIFO structure in [81]. The FIFO depth and delay are configurable, by default 5-flit deep and 3 reading cycle latencies [81]. The delay on the level shifter is negligible compared to the FIFO delay [33], thus omitted.

A.3 Simulating Dynamic Clusterization

The simulator enables run-time clusterization of any nodes in NoC (Fig. A.2). Each router can be configured with a monitor, which is attached with a list as the cluster look-up table (Section 6.2). If a node (router and processing element) is to be assigned to a specific cluster, the corresponding monitor will write its location into the look-up table. In this manner, virtual clusterization is achieved, which can be easily reconfigured. In each router module, there is a variable recording the monitor location.

To enable communication between the monitor and the affiliated network nodes, a dedicated network is built for sending monitoring information and reconfiguration commands (Fig. A.2). The network is much narrower than the data network, considering the monitoring communication is typically lower in volume than the data communication. There are other alternatives for building the monitoring networks, for instance time-multiplexing. The simulator adopts this architecture as it ensures the transmission performance of monitoring communication and is energy efficient [36].

Figure A.2: Cluster Look-up Table and Run-Time Monitoring Communication for Dynamic Clusterization

A.4 Simulating Cluster-based Reconfiguration

With the clusters dynamically configured, run-time monitoring and reconfiguration in each cluster supervised by the monitors can be simulated (Fig. A.3).

In terms of monitoring, the information of local network condition is collected distributedly. In the simulator, counters are added to trace the information such as workload and latency. The information is sent by each router via the monitoring network to the monitor.

In terms of reconfiguration, each router can be dynamically configured with one of the available supply voltages (emulating multiple on-chip power delivery networks; Section 3.2.1) and with a frequency rationally-related to a reference clock. For simulation, the voltage switching only influences the power/energy consumption, which is recomputed if the router is configured with a new voltage.

The frequency switching assumes ratiochronous timing [14], or in particular GRLS (globally ratiochronous and locally synchronous timing). It is a form of GALS (globally asynchronous and locally synchronous timing). The clocks of local components are rationally-related (i.e. in the form of $\frac{M}{N}$, M and N being integers) to a reference clock. Ratiochronous timing is low in area and energy overheads [14], for instance with simple distributed digital frequency dividers [72].

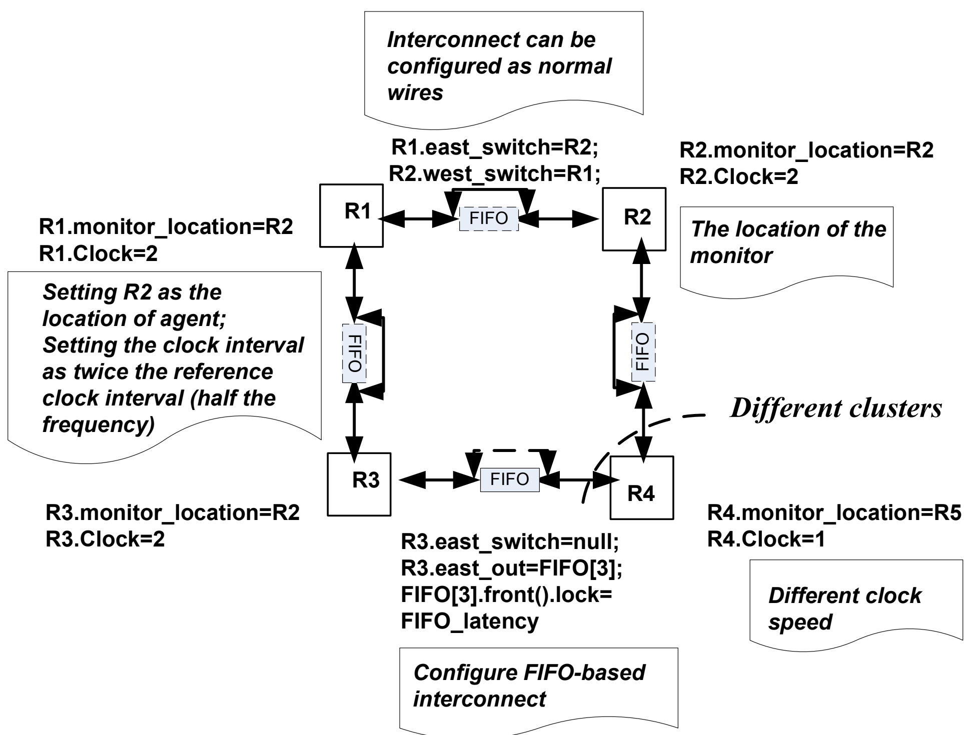


Figure A.3: Simulating Ratiochronous Timing in DoNoC

To simulate synchronous and ratiochronous timing, the iteration construct in sequential programming is utilized. All parallel operations that can be executed in one iteration (without dependencies) are performed before the next iteration starts. Each cycle of the reference clock is an iteration. In one iteration, all parallel operations in the network (the four pipeline stages) are executed. As the same data cannot be operated on twice during one cycle, a lock is added to every newly updated data. The locks are released in the next iteration. With the iteration construct, ratiochronous timing can be simulated. Instead of being updated every iteration, a router is being updated based on its required frequency. For instance, if the router's frequency f_l is half the reference clock frequency f_r, then the router is activated every $\frac{f_r}{f_l} = 2$ iterations. The iteration-based simulation can not be used for handshaking-based asynchronous network.

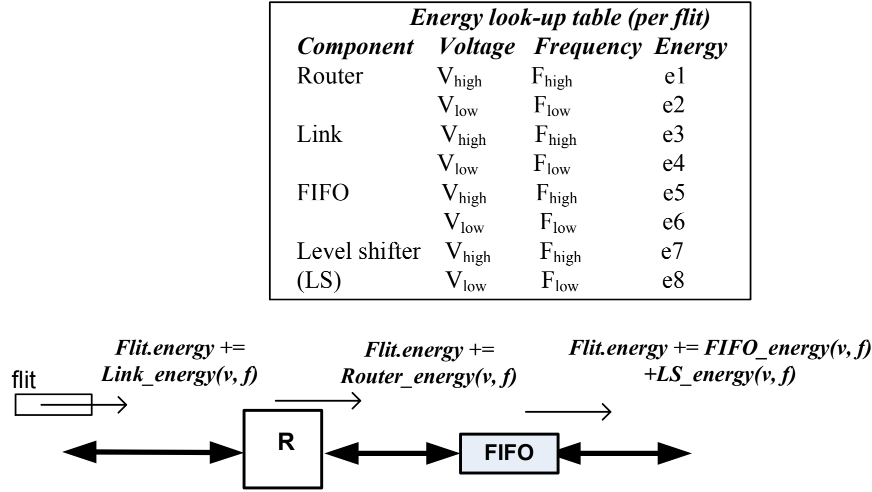
A.5 Integrating Power Models for DoNoCs

The simulator considers the energy consumption of router, link, FIFO-based synchronizer and level shifter. When a flit goes through each of these network components, the traversal energy is added (Fig. A.4). The energy is specified for different voltage and frequency levels in case of run-time power management. The average power can be calculated with the energy divided by the transmission time.

For router and link power, Orion 2.0 [55] is utilized. Given the voltage, frequency, technology and switch architecture parameters, we can obtain the energy consumption. The major energy consumption in FIFO-based synchronizers lies in the shift registers. We estimate the access energy of FIFOs also from Orion 2.0, as input buffers of a router are also modeled with shift registers. As the energy consumption of level shifters strongly depends on the implementation and the output load, the simulator enables reconfiguring the values based on existing literatures (by default using [33]).

A.6 Traffic Generation

The traffic is generated by modules written in Matlab considering its strength in data processing. The message has the format of one flit header followed by a number of payload flits, with the last one being the tail. The traffic generator



Energy look-up table (per flit)

Component	*Voltage*	*Frequency*	*Energy*
Router	V_{high}	F_{high}	e1
	V_{low}	F_{low}	e2
Link	V_{high}	F_{high}	e3
	V_{low}	F_{low}	e4
FIFO	V_{high}	F_{high}	e5
	V_{low}	F_{low}	e6
Level shifter	V_{high}	F_{high}	e7
(LS)	V_{low}	F_{low}	e8

Figure A.4: Integrating Power Models for Run-Time Voltage and Frequency Switching

provides the required packetization, and adds the timing information as the meta-data attached to each flit.

Two categories of traffics are utilized by the simulator, synthetic traces and application traffic traces. The synthetic traces are generated by certain traffic models, which capture the most important features of workloads and are easy to design and modify [26]. Application traces can be extracted from inter-processor communication graphs in practical applications based on existing literature (e.g. [116]).

A.7 Simulator Feature Summary

With the integration of dynamic clusterization and run-time monitoring/reconfiguration functions, the simulator is able to perform system-level design analysis of DoNoC, especially for cluster-based power management, as summarized by Table A.1.

Table A.1: Enhanced Features for Simulating DoNoCs

Category	Features
Conventional NoC functions	Various topologies: mesh, torus, or any random topologies; Switching and routing: wormhole switching or virtual cut-through, X-Y deterministic routing; Traffic: synthetic and practical application traces
Timing	Synchronous, ratiochronous
Run-time monitoring	Load of each router, average load of any clusters (or the whole network); Latency of a particular flit/packet, average latency of all packets; Number of packets received in any time window on any router
Dynamic reconfiguration	Per-core DVFS; DVFS in statically partitioned clusters; Dynamic clusterization; Dynamic cluster based power management; Extension for fault-tolerance
Measurement	Performance of network communication; Power/energy of network communication; Latency/energy overhead for synchronization; Energy overhead of monitoring communication