Network-on-chip based manycore systems

Liao, Xiongfei

2011

Liao, X. F. (2011). Network-on-chip based manycore systems. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/46287>

<https://doi.org/10.32657/10356/46287>

NANYANG TECHNOLOGICAL UNIVERSITY

Network-on-Chip Based Manycore Systems

Xiongfei LIAO

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirements for the degree of

Doctor of Philosophy

2011

Acknowledgments

I would like to take this opportunity to express my appreciation to the following people.

Dr. Thambipillai Srikanthan, my advisor, who is a full professor at the School of Computer Engineering (SCE), the director of Centre for High Performance Embedded Systems (CHiPES) and the Chair of SCE, for his patience, guidance, generosity, and support for the past years. It was a great pleasure and yet humbling experience to work under Dr. Srikanthan, and I have learnt a great deal from observing his philosophy and way of handling both technical and non-technical aspects of life. The Ph.D. training under Dr. Srikanthan makes myself ready for future challenges in my career.

I would like to thank **My Family** for their constant love, encouragement, and support. Special appreciation is given to my family for constantly challenging me to continue my Ph.D study, which served as a much need gravitational force for putting me back on track during the numerous times that I got distracted. Great thanks to my wife, **Ying Zhou**, for her encouragement and love during the hardest times.

Dr. Jigang Wu, who was a research fellow at CHiPES, for the discussions where we shared ideas in the initial stage of my Ph.D study. I especially appreciate his reviews of my work.

Thanks go to my friends at CHiPES, **Dr. Yi Wang, Dr. Aijiao Cui, Dr. Raveendranatha Panicker Mahesh, Dr. Kavallur Pisharath Gopi Smitha, Lam Siew Kei, Jagadeesh R. George, Chua Ngee Tat, Yuan Gu, Sunita Chandrasekaran, Ramya Muralidharan, Sathoda Ravi Kumar, Suchitra Sathyanarayana, Ku Wei Chiet, Navin Michael, Jin Cui, Mathias Faust, Yan Lin Aung, Lin Mengda, Alok Prakash, Amit Kumar Singh, Yupeng Chen, Yongchao Liu, George Rosario Dhinesh, Wu Meiqing, Hanhua Qian, Sharad Sinha, Chen Xuan, Prabhu Kaliamoorthi, Xiwei Huang and Prashob Ramachandran Nair** for the friendly working environment and memorable time during my stay at CHiPES.

Last but not least, I am grateful to **Nanyang Technological University** and the **School of Computer Engineering** for providing financial support and world-class research facilities.

Contents

Acknowledgments	i
List of Figures	viii
List of Tables	xi
List of Acronyms	xii
A List of Publications	xiv
Abstract	xvi
1 Introduction	1
1.1 Background and Motivation	1
1.2 Challenges and Our Contributions	2
1.3 Thesis Organization	5
2 Literature Review	7
2.1 Introduction to Manycore Processors	7
2.1.1 Trends in Semiconductor Technology	7
2.1.2 Utilization Wall	8
2.1.3 Evolution of Multicore Architecture	9
2.1.4 Manycore Processors	10
2.2 Introduction to Network-on-Chip (NoC)	11
2.2.1 Communication Architectures for Manycore Processors	11
2.2.2 Layered Model of Network-on-Chip	11
2.2.3 Topology	13
2.2.4 Routing	13
2.2.5 Flow Control	14
2.2.6 Router Micro-architecture	15

2.2.7	Network Interface	15
2.3	Simulators for Multi-/Many-core Platforms	16
2.3.1	Introduction to Architectural Simulation and Simulators	16
2.3.2	Simulators for Network-on-Chip	17
2.3.3	Acceleration Techniques for Modular Simulators	18
2.4	Resource Management Techniques	19
2.4.1	Runtime Resource Management Techniques	20
2.4.2	Submesh Based Processor Allocation Schemes	21
2.5	Runtime Thermal Management	22
2.5.1	Thermal Problems	22
2.5.2	Runtime Thermal Management Techniques	22
2.6	Conclusion	23
3	Simulation Framework for Network-on-Chip Based Manycore Systems	26
3.1	Introduction and Motivation	26
3.1.1	Contributions and Chapter Organization	27
3.2	Preliminaries	27
3.2.1	UNISIM Environment	27
3.2.2	Cycle-level Modeling in UNISIM	29
3.2.3	Network-on-Chip	30
3.3	Baseline Architecture for NoC-based Manycore Systems	37
3.3.1	A Network-on-Chip and Its General Structure of Tiles	37
3.3.2	Virtual Channel Router	39
3.4	Design and Implementation of the Framework	39
3.4.1	Layers of Software Design	40
3.4.2	Virtual Channel Router and Its Inner Modules	40
3.4.3	Network Interface (NI)	44
3.4.4	Configurable Parameters of the Framework	44
3.4.5	Metrics for Performance Evaluation	45
3.5	Testing the Simulation Framework	46
3.6	Integrating Processor Cores with NoC Infrastructure	48
3.6.1	Microarchitecture	49

3.6.2	Interfacing The PowerPC 405 core with NoC	50
3.6.3	Programming interfaces	53
3.6.4	Evaluation of an MPI based parallel program	56
3.7	Summary	61
3.7.1	Novelty of Our Research	62
4	Accelerating Micro-architectural Simulations on Multicore Platforms	63
4.1	Introduction and Motivation	63
4.1.1	Contributions and Chapter Organization	65
4.2	Motivation for Multithreaded UNISIM Cycle-level Simulation	65
4.2.1	Overview of Single-threaded UNISIM Cycle-level Simulation	66
4.2.2	Exploring Fine-grained Parallelism	67
4.3	The Proposed Systematic Framework	69
4.3.1	Ideas for Multithreaded UNISIM Cycle-Level Simulations	69
4.3.2	The Proposed Framework	70
4.4	Parallelizing The Single-threaded UNISIM SystemC Engine	73
4.4.1	Sequential and Parallel Sections within A Clock Phase	74
4.4.2	Master Thread and Worker Threads in Parallel Simulation	74
4.4.3	UNISIM SystemC Simulation Semantics	75
4.4.4	Performance Optimization and Thread Safety in Multithreaded Simulations	75
4.5	Accelerating Multithreaded Simulations	77
4.5.1	Microcycle	77
4.5.2	Multithreaded Simulation within A Microcycle	77
4.5.3	Factors Affecting Acceleration	78
4.6	Partitioning for Load Balancing	80
4.6.1	Partition Graph of A Simulated System	80
4.6.2	Graph Partitioning Based Technique for Load Balancing	81
4.7	Deploying The Multithreaded Engine for Adaptive Simulations	83
4.7.1	Strategy for Accommodating Computation Variations at Runtime	83
4.7.2	Distributing SystemC Objects for Load Balancing	84
4.7.3	Strategies for Adaptive Multithreaded Simulations	85

4.8	Experiments and Results	87
4.8.1	Experimental Setup	87
4.8.2	Performance Evaluations	89
4.9	Summary	96
4.9.1	Novelty of Our Research	97
5	A Scalable Strategy for Runtime Resource Management	98
5.1	Introduction and Motivation	98
5.1.1	Contributions and Chapter Organization	100
5.2	Preliminaries	101
5.2.1	Overview of Existing Strategies	101
5.2.2	Model of Applications	102
5.3	Submesh-based Resource Management	103
5.3.1	Submeshes for Resource Management	103
5.3.2	Algorithms for Submesh Allocation and Deallocation	104
5.4	The Proposed Hierarchical Strategy	107
5.4.1	Overview	107
5.4.2	Off-line Preprocessing of Applications	108
5.4.3	Hierarchical Resource Allocation	111
5.4.4	Hierarchical Resource Deallocation	112
5.4.5	An Illustrative Example	113
5.5	Evaluation of The Proposed Strategy	114
5.5.1	Experimental Setup	114
5.5.2	Experimental Results	116
5.6	Conclusion	119
5.6.1	Novelty of Our Research	119
6	Hybrid Non-Preemptive/Cooperative Multi-tasking	120
6.1	Introduction and Motivation	120
6.1.1	Contributions and Chapter Organization	122
6.2	Cooperative Multi-tasking for Systems with Multiple CPUs	122
6.2.1	Multi-tasking Approaches for Single-CPU Systems	122

6.2.2	Preemptive Multi-tasking for Systems with Multiple CPUs	123
6.2.3	Cooperative Multi-tasking for Systems with Multiple CPUs	124
6.3	Hybrid Non-preemptive/Cooperative Multi-tasking	125
6.3.1	Overview	125
6.3.2	The Negotiation Mechanism	127
6.3.3	Architectural Supports	129
6.3.4	The Method for Designing Cooperative Applications	131
6.3.5	Discussions	132
6.4	Evaluation of The Proposed Hybrid Multi-tasking	133
6.4.1	Cooperative Application Example: MPEG-2 Encoder	133
6.4.2	Experiments and Experimental Setup	142
6.4.3	Evaluation of The Parallelized MPEG-2 Encoder	144
6.4.4	Experiments for Hybrid Non-preemptive/Cooperative Multi-tasking . .	145
6.4.5	Summary of Experiments for Hybrid Multi-tasking	149
6.5	Conclusion	149
6.5.1	Novelty of Our Research	150
7	Runtime Thermal Management for NoC Based Manycore Systems	151
7.1	Introduction and Motivation	151
7.1.1	Contributions and Chapter Organization	154
7.2	Preliminaries	154
7.2.1	Several Terminologies about Submesh	154
7.2.2	Acquisition of Runtime On-chip Temperature	155
7.3	A Scheme for Temperature-aware Contiguous Submesh Allocation	157
7.3.1	Temperature-aware Allocation and Deallocation Algorithms	157
7.3.2	Temperature-aware Allocation Policies	158
7.3.3	Evaluation of The Proposed Scheme	162
7.4	Temperature-aware Virtual Submeshes	167
7.4.1	Motivational Example	167
7.4.2	A New Form of Virtual Submesh	168
7.4.3	Temperature-aware Virtual Submeshes	170
7.4.4	Algorithm “Maximum-Cut”	171

7.4.5	Algorithm “Minimum-Expand”	174
7.4.6	Comparison of Algorithms	177
7.5	An Adaptive Scheme for Temperature-aware Submesh Allocation	178
7.6	Runtime Thermal Management for NoC based Manycore Systems	180
7.7	Summary	182
7.7.1	Novelty of Our Research	183
8	Conclusions and Future Work	184
8.1	Conclusions	184
8.2	Future Work	186
	References	188

List of Figures

1.1	Thesis Overview	5
2.1	The layered model of Network-on-Chip	12
3.1	Several topologies of Network-on-Chip	30
3.2	Structures of an NoC and its tiles	38
3.3	A typical virtual channel router	39
3.4	Pipeline of the virtual channel router	42
3.5	A simulator for testing the developed simulation framework	47
3.6	An NoC-based homogeneous MPSoC with an 8×8 mesh	50
3.7	Format of added instructions	51
3.8	Root process runs on a tile at corner	57
4.1	Time for simulating each cycle of a simulation on host computer	67
4.2	Overview of the proposed technique	71
4.3	Sequential and parallel subsections in a clock phase	73
4.4	Microcycles in a simulated cycle	77
4.5	Parallel simulations in a microcycle	78
4.6	An example of partition graph, G	81
4.7	Partition graph G with partitions	82
4.8	An NoC-based homogeneous MPSoC with 4×4 mesh	88
4.9	Baseline simulations driven by the single-threaded UNISIM cycle-level engine	90
4.10	Speedups in non-automated adaptive simulations	91
4.11	Impacts of Threshold Imbalance Ratio (TIR)	92
4.12	Impacts of Length Of Period (LOP)	94
4.13	Speedups in fully automated adaptive simulations	95

4.14	Number of worker threads in periods during fully automated simulations	96
5.1	The application characteristics of five applications	99
5.2	Mapping of five applications in Fig. 5.1 to a 5×5 NoC	99
5.3	An NoC with 5×5 2D mesh used in existing strategies	101
5.4	Submeshes on a 64-tile manycore NoC	104
5.5	System configurations at allocation/deallocation stages	106
5.6	Mapping of tasks of applications inside submeshes	110
5.7	A hierarchical mapping of five applications to an 8×8 NoC	111
5.8	Mapping applications onto rectangular regions	113
5.9	An embedded manycore NoC with 5×6 2D mesh	115
5.10	The microarchitecture of a tile	115
5.11	System Configurations at Time 4	118
6.1	Concurrent applications on a 64-tile embedded manycore NoC	121
6.2	An example of hybrid non-preemptive/cooperative multi-tasking	126
6.3	An embedded manycore NoC with a bidirectional control network	130
6.4	The hierarchy of layers in an MPEG-2 bit-stream	134
6.5	The layered structure of MPEG-2 data	135
6.6	The typical dataflow and components of an MPEG-2 encoder	135
6.7	Data parallelism at slice level	137
6.8	Two slices with extra data to avoid increasing bit-rate	138
6.9	Flowcharts for master (a), slave (b) and output (c) processes	139
6.10	The data flows between processes	140
6.11	The sequence of sending/receiving messages between processes	141
6.12	The microarchitecture of a tile	143
6.13	Screen snapshots of the original and the compressed videos	143
6.14	Total number of execution cycles under different configurations	145
6.15	Achieved speedups under different configurations	145
6.16	Speedups for the individual frames in Experiment 1	148
6.17	Speedups for the individual frames in Experiment 2	148
7.1	The allocation algorithm	159

7.2	The deallocation algorithm	160
7.3	Example for allocation policies	161
7.4	NoC architecture and cores	163
7.5	Normalized variances	166
7.6	A Network-on-Chip with six overheated cores	167
7.7	A virtual submesh formed by deleting rows and columns	168
7.8	Two virtual submeshes formed by virtual columns	169
7.9	Virtual submeshes with different thermal features	170
7.10	Construction of a local optimal cool virtual column	173
7.11	Cool virtual submeshes constructed with $Algo_{min}$ (left) and $Algo_{max}$ (right) . .	176
7.12	The allocation algorithm of the adaptive scheme	179
7.13	A technique for runtime thermal management	181

List of Tables

2.1	Research challenges and existing work	25
3.1	High-level wrapper functions	54
3.2	Relevant details of some flits in the simulation	58
3.3	Relevant details of all messages in the simulation	59
3.4	Delay and overheads of all messages in the simulation	60
4.1	Configurations for simulators and T_s	89
4.2	Results of non-automated adaptive simulations	90
4.3	Impacts of Threshold Imbalance Ratio (TIR)	92
4.4	Impacts of Length Of Period (LOP)	93
4.5	Results of fully-automated adaptive simulations	95
5.1	Events in experiment	117
5.2	Execution cycles of allocation algorithms by the Global Manager	117
6.1	Control messages used in negotiation	129
6.2	The average execution time of different functions (in clock cycles)	136
6.3	Different configurations and achieved speedups	144
6.4	Events in experiments	146
6.5	Statistics of processing individual frames in Experiment 1	147
7.1	Policies and their scheduling results	162
7.2	A continuous workload and its tasks	165
7.3	Peak temperatures under policies	166
7.4	Comparing thermal features of virtual submeshes	171
7.5	Comparison of allocated submeshes by $Algo_{min}$ and $Algo_{max}$	177

List of Acronyms

ACG	: Application Communication Graph
API	: Application Programming Interface
CISC	: Complex Instruction Set Computing
CLM	: Cycle Level Modeling
CMOS	: Complementary Metal Oxide Semiconductor
CMP	: Chip MultiProcessor
CMT	: Chip MultiThreading/Multi-Threaded
CPU	: Central Processing Unit
CUDA	: Compute Unified Device Architecture
DSP	: Digital Signal Processing/Processor
DVFS	: Dynamic Voltage and Frequency Scaling
EDA	: Electronic Design Automation
FPGA	: Field-Programmable Gate Array
FSM	: Finite State Machine
GALS	: Globally Asynchronous Locally Synchronous
GCC	: GNU Compiler Collection
GPU	: Graphics Processing Unit
ILP	: Instruction Level Parallelism
ILP	: Integer Linear Programming
IP	: Intellectual Property
ISA	: Instruction Set Architecture
ITRS	: InternaTional Roadmap for Semiconductor
MOSFET	: Metal Oxide Semiconductor Field Effect Transistor
MPEG	: Moving Picture Experts Group
MPI	: Message Passing Interface
MPSoC	: Multiprocessor System-on-Chip
MSSG	: MPEG Software Simulation Group
NI	: Network Interface
NoC	: Network-on-Chip
OS	: Operating System
OSCI	: Open SystemC Initiative
OSI	: Open Systems Interconnection
POSIX	: Portable Operating System Interface for UNIX
QoS	: Quality-of-Service

RISC	:	Reduced Instruction Set Computing
RTOS	:	Real-Time Operating System
SAF	:	Simultaneously Active Fraction
SDF	:	Synchronous Data Flow
SDRAM	:	Synchronous Dynamic Random Access Memory
SMT	:	Simultaneous Multi-Threading
SoC	:	System-on-a-Chip
SPARC	:	Scalable Processor ARChitecture
SPMD	:	Single Program Multiple Data
TDMA	:	Time Division Multiple Access
TLM	:	Transaction Level Modeling
UNISIM	:	UNIted SIMulation environment
VCA	:	Virtual Channel Allocator
VHDL	:	VHSIC Hardware Description Language
VLSI	:	Very Large Scale Integration
XML	:	Extensible Markup Language

A List of Publications

During my Ph.D. candidature, I have published in international conferences and journals. Publications directly related to this thesis are listed as follows.

Journal Publications

- **Xiongfei Liao**, Wu Jigang and Thambipillai Srikanthan. A Modular Simulator Framework for Network-on-Chip based Manycore Chips Using UNISIM, *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC) IV, LNCS 6760*, pp. 237-256, 2011. (Chapter 3)
- **Xiongfei Liao** and Thambipillai Srikanthan. Accelerating UNISIM Based Cycle-Level Microarchitectural Simulations on Multicore Platforms, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol. 16, No. 3, Article 26, May 2011. (Chapter 4)
- **Xiongfei Liao** and Thambipillai Srikanthan. A Scalable Strategy for Run-time Resource Management on Embedded Manycore Networks-on-Chip, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, submitted. (Chapter 5)
- **Xiongfei Liao** and Thambipillai Srikanthan. A Technique for Hybrid Non-preemptive / Cooperative Multi-Tasking on Embedded Manycore Networks-on-Chip, *Journal of Systems Architecture: Embedded Software Design*, submitted. (Chapter 6)
- **Xiongfei Liao** and Thambipillai Srikanthan. Techniques for Runtime Thermal Management on Network-on-Chip Based Manycore Systems, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, to be submitted. (Chapter 7)

Conference Publications

- **Xiongfei Liao**, Thambipillai Srikanthan and Xiao He. Hybrid Non-preemptive/Cooperative Multitasking on NoC Based Manycore Systems, *International Symposium on Integrated Circuits (ISIC 2011)*, 2011. (Chapter 6)
- **Xiongfei Liao** and Thambipillai Srikanthan. A Scalable Strategy for Runtime Resource Management on NoC based Manycore Systems, *International Symposium on Integrated Circuits (ISIC 2011)*, 2011. (Chapter 5)
- **Xiongfei Liao** and Thambipillai Srikanthan. Improving Performance of SystemC Based Cycle-Level Simulations Using Multicore Platforms, poster at *the International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*, 2009. (Chapter 4)
- **Xiongfei Liao**, Wu Jigang and Thambipillai Srikanthan. A Modular Simulator Framework for Network-on-Chip based Manycore Chips, *IEEE/ACM Workshop on Software and Hardware Challenges of Manycore Platforms (SHCMP'08) in conjunction with the 35th International Symposium on Computer Architecture (ISCA'08)*, 2008. (Chapter 3)
- **Xiongfei Liao**, Wu Jigang and Thambipillai Srikanthan. A Temperature-aware Virtual Submesh Allocation Scheme for NoC-based Manycore Chips, *the 20th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'08)*, 2008. (Chapter 7)
- **Xiongfei Liao**, Wu Jigang and Thambipillai Srikanthan. Temperature-Aware Submesh Allocation Scheme for Heat Balancing on Chip-Multiprocessors, *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'07)*, 2007. (Chapter 7)

Abstract

As deep sub-micron technologies advance, architectures of microprocessors have evolved from traditional monolithic ones into parallel ones which consist of large number of small but energy efficient cores that rely on Network-on-Chip (NoC) communication infrastructure to achieve scalability.

This thesis presents the design and development of a comprehensive simulation framework for an NoC based system simulator covering all system levels from low-level hardware modules to high-level software applications. The motivation stems from the fact that there exists no publicly available simulator that supports embedded NoC based manycore systems.

A cycle-level simulation framework has been proposed using the UNISIM environment to complement the configurable on-chip network built upon efficient pipelined routers that employ wormhole switching and virtual-channel flow control. In order to accelerate the cycle-level micro-architectural simulations for manycore systems, novel techniques have been devised to accelerate the cycle-level simulations on multi-core platforms. In particular, we have exploited the fine-grained parallelism within each simulated cycle using Pthreads, leading to notable speedups. The proposed multithreaded simulation engine exploits inherent parallelism with the help of an adaptive technique for managing the computation workload and relies on a graph partitioning based technique for automating load balancing among workloads of multi-threaded executions. Our investigations show that, by adaptive distribution of modules among multiple CPU cores of the simulation platform at runtime, the proposed techniques can provide for up to 6X speed up using an 8-core computer.

Existing allocation strategies do not lend well NoC based many-core systems when their core counts increase. In this thesis, a runtime resource management strategy has been proposed to overcome these limitations thereby making it suitable for NoC based manycore systems that must support a large number of cores. The proposed resource management strategy relies on submesh based scheme for organizing resources at runtime allocation/deallocation process. It

employs a hierarchical-based resource allocation and deallocation scheme to speed-up and to save energy.

A hybrid scheme based on non-preemptive and cooperative multi-tasking has been proposed to attain high flexibility and to overcome the inherent limitations of the existing methods. Methods for parallelizing applications and for negotiating between the system and applications with the help of architectural support have also been devised to support the proposed multi-tasking scheme. Finally, the proposed hybrid multi-tasking technique has been demonstrated with the help of an MPEG-2 encoder application.

Thermal-aware schemes have been devised in this thesis to enhance reliability. In particular, novel techniques for thermal-aware resource allocation and a solution for runtime thermal management by combining the allocations of thermal-aware submeshes and runtime task migration have been proposed. In particular, a contiguous scheme for temperature-aware submesh allocation has been proposed to identify and provide thermal-aware resources for applications at runtime. In addition, an efficient scheme based on virtual submesh has been devised to manage overheated cores by treating them as temporarily faulty. Two fast heuristics have been also devised to dynamically construct virtual submeshes in a thermal-aware manner. The proposed temperature-aware task migration scheme ensures that the redundant cores and those temporarily faulty cores that have cooled down can be re-employed at runtime.

The proposed techniques lend well for constructing an efficient simulation environment and for realizing runtime resource management for NoC based manycore systems. The emphasis on scalability and reliability throughout the various contributions has made it possible to propose novel solutions. The thesis concludes with suggestions for future research efforts in this fast emerging field.

Chapter 1

Introduction

1.1 Background and Motivation

Following Moore’s Law [Moo07], billions of transistors can pack into a single chip and thereby one of the industry’s trends has been towards multicore architectures due to the so called “power wall” [AHKB00], “ILP wall” [Wal91] and “memory wall” [WM95] [McK04]. Major vendors have stopped the relentless pursuit of individual CPU performance and have instead been doubling the number of CPU cores per chip with each generation. Multicore processors are not only the mainstream in server and desktop domains, but also entering small devices including laptops, tablets and handhelds. Current multicore processors commonly have less than a dozen cores and examples are Intel’s Nehalem and Xeon families, AMD’s Opteron and Phenom families, SUN’s Niagara and IBM’s Power 7.

Following the above trend, it was acknowledged that the manycore era was fast approaching [ABC⁺06] [Bor07]. A typical manycore processor may have hundreds, even thousands of relatively small CPU cores and other uncore components (the parts of a processor that are not the CPU cores, such as L3 cache, the on-die memory controller, and other bus controllers) on a single die. The resources are suggested to be connected by Network-on-Chip (NoC). It has been shown that NoC architecture overcomes the limits by wire delay, reduces design complexity and power consumption, and hence enables scalability of designs [BD02].

Realizations of *NoC based homogeneous manycore processors* started to appear. Two example processors with 2D mesh topology are Intel’s 80-tile chip [VHR⁺07] and Tilera’s 100-core TILE-Gx100 [Til09]. The 2D mesh is popular because it is regular, simple and predictably scalable with regard to power and area [BD06]. These homogeneous processors (all tiles are

identical) are considered as important high performance platforms for general purpose (or multiple use-cases) computing and thereby have been taken as the target platforms in many research works [COM08] [TCR⁺09] [CM10]. We are particularly interested in NoC based homogeneous manycore processors suitable for embedded domain and these processors are called “*embedded manycore NoCs*” in this thesis for brevity.

Architectural simulators have been widely used for evaluating different system designs without building costly physical hardware systems and for obtaining detailed performance metrics [SMA⁺03] [HP06]. In this thesis research, we adopt architectural simulation as the research methodology. Moreover, we prefer simulators that are open-source and can run on the general-purpose CPU based platforms such that they can be available to most researchers.

1.2 Challenges and Our Contributions

There are several challenges for embedded manycore NoCs, which will be addressed in this thesis research, as described below.

- Investigation on architectural simulators presented in literature shows that simulators proposed for NoC related research are not suitable for embedded manycore NoCs due to below reasons. 1) Some simulators cannot support coupled simulations of CPU cores and NoC. Examples include NNSE [ZLJ05], Noxim [Nox08] and NIRGAM [Nir07]. 2) The CPU models of some simulators are not appropriate for embedded manycore NoCs. For instance, the combination of GEMS [MSB⁺05] and GARNET [APJ08] can support coupled simulations of cores and NoCs. However, the aggressive out-of-order CPU models used in GEMS are not appropriate for embedded manycore NoCs where smaller and energy efficient CPU cores are desired [ABC⁺06]. Therefore, architectural simulators suitable for embedded manycore NoCs have to be developed.
- As embedded manycore NoCs have abundant cores and uncores, cycle-level architectural simulations of these NoCs are envisioned to be slow. Thus, acceleration techniques are highly desired. Nowadays, multicore computers are prevalent for speedup through on-chip, thread-level parallelism. The simulators to be developed must utilize the parallel processing capability of the multicore computers in order to achieve high performance.

However, cycle-level simulations are difficult to parallelize as they are essentially sequential and can only be advanced cycle by cycle. In addition, cycle-level architectural simulators are coded in a way to imitate the architecture-level activities, and commonly don't allow their source code to be modified. These present challenges for the desired acceleration techniques.

- Embedded manycore NoCs can execute several applications concurrently and thus lead to dynamic system configurations in the use of CPU cores. As such configurations are extremely difficult to model off-line, run-time techniques are indispensable. Existing strategies, like [COM08] and [CM10], have been proposed for resource allocation. However, the scalability of these strategies is limited due to their adopted centralized resource management and communication contentions among tasks. Moreover, due to the adopted non-preemptive multi-tasking, after resource allocation is completed, the interactions among applications are not allowed. Suitable techniques need to be explored for scalable and flexible resource management.
- The thermal problems due to high power consumption at runtime are critical for the reliability of manycore processors because all cores of such a processor share a same small die and there is no separate cooling device for individual cores. Unfortunately, suitable thermal management techniques have not been proposed for manycore processors in the literature.

In this thesis research, we seek to tackle the above challenges by 1) developing modelling, simulation and acceleration techniques, and 2) exploring techniques for runtime management. We summarize our main research contributions as follows.

- *Simulating NoC-based Manycore Systems*

A simulation framework based on a modular infrastructure, i.e., UNISIM environment [ACG⁺07], has been developed for generating architectural simulators that support small and energy-efficient CPU cores connected by advanced NoCs. These architectural simulators enable cycle-level coupled simulations of CPU cores and NoC. Particularly, an example simulator for embedded manycore NoCs, incorporating the PowerPC 405 core, has been generated to support message-passing programming model and it can run parallel applications written with a subset of MPI APIs.

- *Accelerating Micro-architectural Simulations on Multicore Platforms*

A systematic technique has been developed to accelerate cycle-level architectural simulators based on aforesaid framework on multicore platforms by exploiting thread-level parallelism. This technique is applied at simulation engine level and doesn't require any modification to source code of simulators. In addition, this technique can be generalized to be applied to any discrete event simulation engine with delta-delay semantics.

Particularly, the UNISIM SystemC engine is parallelized by using Pthreads to exploit fine-grained parallelism within simulated cycles. SystemC modules of a simulator can be divided into disjoint partitions and simulation within a partition is run by a dedicated thread. The original sequential simulation of modules thereby are carried out by multiple threads. To counter variations in computations, workloads of threads are periodically monitored and high performance is achieved by balancing workloads.

- *A Scalable Strategy for Runtime Resource Management*

A scalable hierarchical strategy for runtime resource management has been proposed to overcome the scalability limitation of existing strategies. Our strategy uses submeshes to organize resources and allocates resources to applications in forms of submeshes so as to avoid external communication contentions. In addition, our strategy handles resource management in a hierarchical way to overcome the limitation from centralized resource management. First, a scalable scheme is adopted to manage submeshes for applications. Then, the resources within submeshes are managed in a distributed manner.

- *A Technique for Hybrid Non-preemptive/Cooperative Multi-tasking*

To overcome the limitations resulted from the non-preemptive multi-tasking, a hybrid multi-tasking technique has been proposed to enable interactions among applications after resources are allocated. First, the *cooperative multi-tasking* for systems with multiple CPU cores is introduced. A *cooperative application*, which supports cooperative multi-tasking, can cooperate with the OS at runtime on resource management. Further, as the non-preemptive and cooperative applications can co-exist in the system at the runtime, a *hybrid non-preemptive/cooperative multi-tasking* technique has been proposed to enable interactions between applications.

- *Runtime Thermal Management on NoC-based Manycore Systems*

To keep good heat balance throughout cores at runtime so as to avoid thermal crises [Bor05], a scheme for temperature-aware contiguous submesh allocation is first proposed where a proactive way is adopted to include thermally favourable cores in forms of submeshes when resources are allocated. Moreover, when overheated or faulty cores appear, a new form of virtual submesh, i.e., “temperature-aware virtual submeshes”, is introduced. Furthermore, unlike the existing work which only adopts a reactive strategy for eliminating thermal emergencies through monitoring and control, an adaptive scheme for temperature-aware submesh allocation and a solution adopting combined proactive and reactive strategies are proposed for runtime thermal management.

1.3 Thesis Organization

Figure 1.1: Thesis Overview

As shown in Figure 1.1, the major chapters of this thesis can be broadly partitioned into two parts. The first part lays the foundation for the second part. The first part, consisting of Chapters 3 and 4, presents modelling and simulation of NoC based manycore systems which

are followed by an acceleration technique for cycle-level microarchitectural simulations. The second part explores the runtime management on embedded manycore NoCs. Chapters 5 and 6 investigate the resource management strategies on embedded manycore NoCs by utilizing the simulators developed in the first part. Chapter 7 explores the runtime thermal management on NoC based manycore systems.

The rest of the thesis is summarized as follows.

- Chapter 2 presents a literature review which covers relevant existing works and emerging trends in multi-/many-core platforms, Network-on-Chips, simulation techniques, resource management and thermal management.
- Chapter 3 describes the development of modular cycle-level architectural simulation framework and an example simulator constructed for embedded manycore NoCs.
- Chapter 4 discusses a novel technique for accelerating cycle-level micro-architectural simulations of manycore systems on multicore platforms.
- Chapter 5 presents a scalable hierarchical strategy for runtime resource management on embedded manycore NoCs which overcomes the scalability limitation of existing strategies.
- Chapter 6 discusses a hybrid non-preemptive/cooperative multi-tasking technique for offering interactions among applications at runtime after resource allocation so as to improve flexibility and efficiency of resource usage.
- Chapter 7 presents runtime thermal management techniques on NoC based manycore systems, which are based on the “temperature-aware submeshes”.
- In Chapter 8, conclusions are discussed, followed by some directions for future work.

Chapter 2

Literature Review

This chapter first provides the necessary background of Network-on-Chip based manycore systems, which has been divided into two parts: manycore processors and Network-on-Chip. Then, previous work related to the research of this thesis, which includes architectural simulation, run-time resource management and runtime thermal management, is surveyed.

2.1 Introduction to Manycore Processors

2.1.1 Trends in Semiconductor Technology

The continuous and systematic increase in transistor density and performance, guided by CMOS scaling theory and Moore's Law [Moo07], has been a highly successful process for the development of silicon technology for decades. As alternative techniques are far from application in large-scale electronic devices [Che06], CMOS is likely to be the primary technology for electronics for at least the upcoming decade and there will be more and more transistors in the upcoming devices [ITR06].

However, as the silicon technology scales down to the 45 nm node and beyond, there will be several challenges for CMOS VLSI systems: power, energy, variability, and reliability [Bor05]. Single-event upsets (soft errors) and device (transistor performance) degradation will become more frequent and serious, which make the transistors less reliable.

Moreover, there are issues about wires which are critical for VLSI designs in general [ITR06]. While the speed of gates becomes much faster following the scaling, the wire delay is growing exponentially because of the increased capacitance caused by narrower channel width and increased crosstalk. Besides the wire delay, the wire models are unreliable due to issues like

fabrication variations, crosstalk, noise sensitivity etc. The magnitude of the wiring problem is examined in [HMH01]. Wires that shorten in length as technologies scale have delays that either track gate delays or grow slowly relative to gate delays. Global wires, relatively long wires, do not scale in length since they communicate signals across the chip. The delay of these wires will remain constant if repeaters are used, meaning that relative to gate delays, their delays scale upwards. In future technologies, short wires are preferred and long wires should be avoided.

2.1.2 Utilization Wall

As sufficient reduction in supply voltage cannot be achieved, the relative increase in energy efficiency of individual transistors is lagging behind the growth of their aggregate volume [Bor07]. The power dissipation of a chip is already reaching the practical limits for cost-effective cooling and the power density of hardware components continues to increase when process node scales down [Bor07]. To remain within a reasonable power envelope, the portion of on-chip transistors that can be active simultaneously will have to continue to decrease. *Simultaneously Active Fraction* (SAF) is introduced in [CWS07] to describe the above portion.

A quantitative analysis of the SAF is carried out in [CWS07]. The SAF is computed at each technology node from $90nm$ to $32nm$ as the fraction of the aggregate devices that account for a given target power envelope which is assumed to remain constant across these technology generations. It was found that despite various parameters and design issues there is a downward trend of the SAF if there is no dramatic alteration of device properties.

The above limitation of the fraction of the on-chip resources that can be used at full speed at one time is considered as a technology-imposed “utilization wall” [VSG⁺10], which is concluded as a consequence of CMOS scaling theory combined with modern technology constraints. It is found that under certain power budget, say $80W$, only 17.6%, 6.5% and 3.3% of resources can be run at full frequency in TSMC process at $90nm$, $45nm$ and $32nm$ technology node respectively. Hence, the utilization ratio for resources is quite low for future devices.

The effects of the utilization wall have been already indirectly shown in recent designs. Intel’s Nehalem provides a “turbo mode” that powers off some cores in order to run others at higher speeds [VSG⁺10]. As well, Intel’s Polaris [VHR⁺07] has been designed in a way that certain hardware components can be turned off to realize fine-grained power management.

2.1.3 Evolution of Multicore Architecture

For a very long period, the driving force for computing industry was the relentless pursuit of individual CPU performance by increasing the operating frequency and the number of issued instructions. The *superscalar* processor architecture implements a form of parallelism called *Instruction-Level Parallelism* (ILP) within a single processor and allows faster CPU throughput than would otherwise be possible at the same clock rate [HP06]. Intel's microprocessors have implemented a CISC instruction set on a superscalar RISC micro-architecture [HP06].

However, the achievable performance growth of superscalar microarchitectures slows substantially due to the "memory wall" [WM95] [McK04], "ILP wall" [Wal91], the "power wall" [AHKB00], diminishing improvements in clock rates and poor wire scaling as process shrinks.

So, the trend in utilizing the abundant transistors on a die was changed [HP06]. A single chip is designed to contain several CPU cores and even a whole system, thereby called "multicore". A multicore processor does not necessarily run as fast as the highest performing single-core processors, but it improves overall performance by handling more work in parallel on multiple cooler-running, more energy-efficient processing cores. Industry has adopted the roadmap of making multicore processors [Gee05]. There are many categories of multicore chips for different domains. For server and desktop domains, there are Chip MultiProcessor (CMP) and Chip Multithreading (CMT). Multiprocessor System-on-Chip (MPSoC) is for embedded domain.

After examining the superscalar architecture [SBV95] and the Simultaneous Multi-Threading (SMT) architecture [TEL95], Olukotun et al. proposed CMP architecture [ONH⁺96] [HNO97]. The proposed CMPs use several relatively simple single-thread CPU cores to exploit only moderate amounts of parallelism within any one thread, while executing multiple threads in parallel across multiple CPU cores, which are extremely successful in the server market where there is plenty of process-level or thread-level parallelism.

CMT processors were proposed to support many simultaneous hardware threads of execution via combined features from CMP and SMT, where each core supports multiple hardware threads. The CMT architecture was accepted by industry quickly [TCC⁺00] [Moo00] [Mar03] [MB04] [KST04a] [Dev04] [SA05] [Ope07]. CMT design style gradually spreads to desktop processors [Ote04]. Today, most desktop processors are designed as CMT.

MPSoC platforms, such as PNX-8500 [DJR01] and TI OMAP [Wol04], are introduced to design systems that require multiple heterogeneous, flexible processing elements, a memory hi-

erarchy and I/O components. MPSoCs meet the performance needs of applications in areas such as multimedia, telecommunication, network security while limiting the power consumption.

Multicore processors have major advantages over single-core processors for hardware design [OH05]. They require only a fairly modest engineering effort for each generation. Each member of a family of processors just requires additional copies of the core and making modifications to communication logic connecting the cores together to accommodate the additional cores in each generation. There is no need to completely redesign the core logic. Moreover, the system board design typically needs only minor updates between generations.

2.1.4 Manycore Processors

Processors with a large number of cores, called “manycore” processors [ABC⁺06] [Bor07], are emerging due to the fairly modest engineering effort mentioned above. Azul Systems [Sys10] provides Vega 3 processors with 54 cores for executing Java programs. Intel released “Polaris” which has 80 cores [VHR⁺07]. Tiler provides TILE-Gx100 [Til09] which has 100 cores for embedded domain. It is commonly believed that the “manycore era” is approaching [ABC⁺06] [Bor07] where manycore systems will soon become ubiquitous and predominant, not only in server and desktop machines but also in small client devices.

People believe that there are advantages to building manycore processors with smaller cores, rather than the aggressive, out-of-order cores in current multicore processors [ABC⁺06]:

- 1) Parallelism is an energy-efficient way to enhance performance.
- 2) Many small cores give the highest performance per unit area for parallel code.
- 3) A larger number of smaller processing elements allows a finer-grained ability to perform dynamic voltage scaling and power down.
- 4) A small processing element is an economical element that is easy to shut down in the face of catastrophic defects and easier to reconfigure in the face of large parametric variation.
- 5) A small processing element with a simple architecture is easier to design and functionally verify. In particular, it is more amenable to formal verification techniques than complex architectures with out-of-order execution.
- 6) Smaller hardware modules are individually more power efficient and their performance and power characteristics are easier to predict within existing electronic design-automation design flows.

As there will be more transistors and cores will be smaller, the number of cores on a future manycore processor could be more than 1,000 [Bor07]. A natural thing to consider is the on-

chip communication architecture suitable for such processors with that huge number of cores. We discuss communication architectures for manycore processors below.

2.2 Introduction to Network-on-Chip (NoC)

2.2.1 Communication Architectures for Manycore Processors

With CMOS technologies, the on-chip communication architectures are closely related to the performance of wires. As mentioned in Section 2.1.1, short wires are preferred and long wires should be avoided in future technologies.

Advanced bus architectures have been used on the current multicore processors. These solutions include AMBA [Ltd10], CoreConnect [IBM10] and WISHBONE [Ope10]. These architectures use hierarchical structures to obtain scalable communication throughput and partition communication domains into several groups of communication layers based on bandwidth requirement. However, different bus architectures require modifications in bus implementations and bus interfaces which have negative impacts on reuse of IP cores [BM06a]. Therefore, these advanced bus architectures cannot easily adapt to changes in the system architecture.

Network-on-Chip (NoC) architecture [BD02] has been proposed and shown as a promising communication architecture for manycore processors. Within NoC architectures, routers or switches are inserted between communication nodes which contain IP cores and short wires can be used between various components. In this way, NoC architectures avoid the needs of long wires, complex routing of wires and lead to scalability.

Other benefits of NoC are as follows [DT01] [BD02] [HWC04]. NoC architectures overcome the challenges of signal integrity of wires. Moreover, the most likely synchronization paradigm in deep submicron technologies is globally-asynchronous locally-synchronous (GALS), with many different clocks. NoC can be easily implemented as GALS structure. Furthermore, as NoC scales almost linearly with system complexity, it helps to improve design productivity. Actually, manycore processors mentioned in Section 2.1.4 have adopt NoCs.

2.2.2 Layered Model of Network-on-Chip

An NoC is the physical structure over which different cores of a processor communicate [BD02]. Researchers have adapted a model of layered network communication for NoC which is simplified from the well-known OSI model [BD02] [BM06c]. With this model, the various research

activities can be categorized into different research areas according to these layers. Figure 2.1 shows such a model of NoC. The research activities of NoC can be divided into the following levels: system, network interface, network and link.

Figure 2.1: The layered model of Network-on-Chip

A communication between a pair of cores is performed through transfer of discrete messages between the cores where data messages are further split into *packets*. Packets are composed of one or more *flits*, or *flow control units*. If a packet consists of multiple flits, the first is called the *header*, which contains information which the routers of NoC use to determine where to route the flits of the packet. The last flit is the *tail* flit, which indicates the end of the packet. And the intermediate flits (if any) are *body* flits which contain the actual data being transferred to the destination core. In some NoCs, flits may in turn consists of multiple *phits*, or *physical units*. A phit is the largest number of bits which can be transferred over a physical link between adjacent nodes in a single cycle. A flit is the smallest logical unit of data in the context of NoC.

NoC itself consists of a number of *routers*, or *nodes* which are connected by *physical links*, over which the communication takes place. There are *network interfaces* or *network adaptors* which connect between the CPU cores to routers. The microarchitectures of routers can be different according to the desired features of the NoC.

At the network interface and network levels, an NoC has the following important elements: (1) its topology which decides the connection of components; (2) the routing algorithm which

governs the paths that individual packets take from one processor to another; (3) the flow control protocol which manages network resources such as buffers and switches; (4) the router micro-architecture; (5) network interface which connects the router and components inside a tile.

In the following sections, the basics of the above elements are introduced. More thorough treatments of NoC can be found in books such as *Interconnection Networks: An Engineering Approach* [DYL02], *Principles and Practices of Interconnection Networks* [DT03], *Networks on Chips: Technology and Tools* [BM06a] and *On-Chip Networks* [PJ09].

2.2.3 Topology

The topology of a network is the specific way in which the nodes and links are connected [PJ09]. For NoCs, the topology determines the physical layout and connections between nodes and channels. Several metrics, such as degree, hop count, maximum channel load, path diversity, have been commonly used in comparing topologies at the early stages of design. The topology of NoC has profound effects on the overall cost and performance of the on-chip network. The implementation complexity cost of a topology depends on the following factors: 1) the degree of nodes and 2) the ease of laying out a topology on a chip, which are decided by wire lengths and the number of metal layers required.

The popular topology of on-chip interconnect network is *2D mesh* because it is regular, simple and predictably scalable with regard to power and area [BD06]. Several processors have adopted the mesh topology. The *IMesh* technique used in the processors of Tiler [WGH⁺07] connects all tiles of a processor with five 2D mesh networks, each specialized for a different use. The Intel's research prototype chip Polaris [VHR⁺07] has its 80 tiles arranged as a 10×8 2D mesh [HVS⁺07]. The AsAP (Asynchronous Array of simple Processors) manycore platform for DSP applications, a processor composed of many small and simple processing elements, connects its tiles with a nearest-neighbor mesh interconnect [BYM⁺07].

Beyond the mesh topology, researchers explore other topologies: *concentrated mesh topology* [BD06], *flattened butterfly topology* [KDA07] [KBD07], *the dragonfly topology* [KDSA08], *Fat H-Tree topology* [MKY⁺09] and *ring topology* [KK09a].

2.2.4 Routing

The routing algorithm defines what paths of links and routers individual packets take when travelling from their sources to their destinations [PJ09]. The routing algorithm is used to

distribute traffic evenly among the paths supplied by the topology, so as to avoid hotspots and minimize contention, thus improving network latency and throughput.

Routing algorithms have been proposed before the advent of NoC and many of them can be applied to NoC directly or after minor revision. A deadlock-free routing algorithm can be generated for arbitrary interconnection networks using virtual channels when there are no cycles in a channel dependency graph [DS87]. Without adding physical or virtual channels to network topologies, wormhole routing algorithms are designed to be deadlock free, livelock free, minimal or non-minimal, and maximally adaptive [GN92] [GN94]. The theoretical background for the design such algorithms was developed in [Dua93].

After NoC is introduced, certain new routing algorithms are developed via combining features of existing algorithms. DyAD adaptive routing [HM04] combines the advantages of both deterministic and adaptive routing schemes and judiciously switches between deterministic and adaptive routing based on the network's congestion conditions. The routing algorithm in [WSJZ09] combines the shortest path routing and adaptive routing schemes for NoCs.

Heterogeneity, fabrication faults and reliability may lead to irregular topologies. A region-based routing mechanism is proposed in [FMLD07] [MPF⁺09] that avoids the scalability problems of table-based solutions for irregular topologies. For designing NoC incorporating cores larger than the tile size, two deadlock free routing algorithms for mesh NoC with regions are proposed in [HPK08]. [HKPM09] proposes a methodology for design of deadlock free routing algorithms for hierarchical networks, by combining routing algorithms of component subnets.

2.2.5 Flow Control

Flow control decides the allocation of network buffers and links [PJ09]. It determines when buffers of routers and links are assigned to messages, the granularity at which they are allocated: granularity at which it operates: either at the flit, packet-level, or message-level, and how these resources are shared among the many messages using the network. In determining the rate at which packets access buffers and traverse links, flow control is instrumental in determining network energy and power consumption.

Virtual channel flow control [Dal92] has been widely adopted flow control mechanism for network. After NoC architectures are proposed as a scalable solution to on-chip communication problem, new flow control mechanisms have been proposed. A predictive closed-loop flow

control mechanism [OM06] [OM08] is presented to overcome the shortcomings of flow control algorithms in macronetworks, such as relying on local information, large communication overhead and unpredictable delays, where the congestions are predicted and controlled based on traffic source and router models specifically targeted to NoCs. A novel flow control mechanism, express virtual channels (EVCs) [KPKJ07], is proposed to allow packets to virtually bypass intermediate routers along their path in a completely non-speculative fashion, thereby lowering the energy/delay towards that of a dedicated wire while simultaneously approaching ideal throughput with a practical design suitable for on-chip networks.

2.2.6 Router Micro-architecture

The microarchitecture of router is in large part determined by the network's topology, routing algorithms and flow control [PJ09]. In turn, the microarchitecture determines the power and performance characteristics of the NoC. A router's architecture determines its critical path delay which affects performance including per-hop delay and overall network latency. Router microarchitecture also impacts network energy as it determines the circuit components in a router and their activity. The realization of the routing, flow control and the router pipeline will affect the efficiency at which buffers and links are used and thus overall network throughput. A typical router's microarchitecture consists of input and output channels, routing logic, virtual channel allocation logic, switch allocator, a centralized crossbar, and input buffers.

Many router architectures have been proposed for on-chip networks. After the analysis based on a router delay model that accurately models key aspects of modern routers, a speculative virtual-channel router is proposed in [PD01] that significantly reduces its router latency to that of a wormhole router. A two-stage pipelined router architecture that utilizes adaptive routing while maintaining low latency is proposed in [KPT⁺05] to effectively balance the performance and energy of NoC designs. A fast router is presented in [KKS⁺07].

2.2.7 Network Interface

Network Interface (NI) is considered as the glue logic necessary to adapt the components inside a tile such as compute cores to the router of NoC. Services provided by NI can be classified into the following categories: core adaptation, clock adaptation, network and functional [BM06a].

Many research works have been carried out focusing on issues of network interface. To reduce the latency caused by network interface, the packetization strategies, including software

library based, on-core module based and wrapper based, are examined in [BM03] and the costs in terms of latency, and area are projected through actual synthesis. A Core Network Interface (CNI) architecture is proposed [BM06b] to provide predictable end-to-end latency besides packetizing communication requests and responses. A network interface, which handles the resynchronization between the synchronous and asynchronous NoC domains and implements communication priorities, is proposed in [BV06].

A generic NI architecture and associated wrappers is proposed in [LBYB08] for a networked processor array in order to allow systematic design flow for accelerating the design cycle. The MAIA framework [OMP⁺05] is proposed as a tool for automated NoC generation, automated production of NoC-IP core interfaces and seamless analysis of NoC traffic parameters.

2.3 Simulators for Multi-/Many-core Platforms

Architectural simulators have been widely used for evaluating different hardware designs without building costly physical hardware systems and for obtaining detailed performance metrics [HP06]. They also provide the opportunities to access non-existing computer components or systems and thereby encourage future enhancements and novel systems. Therefore, architectural simulation has been an important research methodology and the majority of research papers published are simulation based [YEL⁺06] [SMA⁺03].

2.3.1 Introduction to Architectural Simulation and Simulators

Based on the modeled scope, simulators can be classified into micro-architecture simulators and full-system simulators [YEL⁺06] [SMA⁺03]. There is a long list of simulators at the website [HYX⁺10]. Some widely used micro-architecture simulators include SESC [RFT⁺05], SimpleScalar [BA97] and SimpleScalar's variants [SNKB01] [DM06a]. Full-system simulators include Simics [MCE⁺02], M5 [BDH⁺06] [Mic07] and GEMS [MSB⁺05] [Wis07].

Because the ideal simulators have two desired features: accuracy and flexibility [YEL⁺06], developers of the current generation of simulators not only try to keep accuracy by modeling the processor in great detail but also focus on flexibility. However, as the modern computing systems are becoming larger and more complex, computer architects must explore relatively large design and application spaces. The detailed modeling and flexibility together pose problems that limit this design space exploration.

Researchers keep seeking solutions to ensure that the above conflicting issues do not inhibit research and design of future computer architecture [YEL⁺06]. Analytical models, such as [KS04], [SOC04], [BVC04], [MAS⁺05], [NC09] and [LGXW09], are proposed to efficiently explore the large design space during the early design phases. Analytical modeling involves developing a limited number of formulas that summarize performance based on program characteristics and microarchitectural parameters.

Statistical simulation combines analytical modeling and simulation to generate a synthetic trace based on program characteristics and is subsequently simulated on a simple trace-driven simulator. Statistical simulators are presented in [OCF00], [NS01], [EBS⁺04] and [BM09] .

A new interesting avenue to explore is the modular simulation infrastructures. The use of these infrastructures such as Liberty Simulation Environment (LSE) [VVP⁺02] [VVP⁺06] and UNISIM [ACG⁺07] [UNI09] allows for the quick building of simulators by assembling several reusable components.

For example, UNISIM is designed to rationalize simulator development by making it possible and efficient to distribute the overall effort over groups, even without direct cooperation. UNISIM achieves this goal with a combination of modular software development, distributed communication protocols, multilevel abstract modeling, interoperability capabilities, a set of services APIs, and an open library/repository for providing a set of simulator modules. The services provide a flexible way to increase the functionality of simulators and to leverage other techniques such as the aforementioned analytical models and statistical simulation.

2.3.2 Simulators for Network-on-Chip

Several simulators have been proposed for study of NoCs.

The Nostrum NoC Simulation Environment (NNSE) [ZLJ05] was developed to enable the analysis of the performance impact of the configuration parameters of on-chip networks. It allows users to: 1) configure a network with respect to topology, flow control and routing algorithm etc.; 2) configure various regular and application specific traffic patterns; 3) evaluate the network with the traffic patterns in terms of latency and throughput.

The Noxim simulator [Nox08] is developed using SystemC. Noxim has a command line interface for defining several parameters of an NoC such as the network size, buffer size, packet

size distribution, routing algorithm, selection strategy, packet injection rate, traffic time distribution, traffic pattern and hot-spot traffic distribution. The simulator allows evaluation of NoC in terms of throughput, delay and power consumption.

NIRGAM [Nir07] is a cycle accurate simulator developed with SystemC for NoC. It allows designers to experiment with various options of NoC: topology, switching technique, virtual channels, buffer parameters, routing mechanism and applications. Besides built-in capabilities, it can be easily extended to include new applications and routing algorithms. It can output performance metrics (latency and throughput) for a given set of choices.

A power-performance interconnection network simulator, Orion [WZPM02], was developed based on architectural-level parameterized power models to provide detailed dynamic power characteristics and performance characteristics, enabling power-performance tradeoffs at the architectural-level. An architectural leakage power modeling methodology [CP03] was proposed to complement the dynamic power models of Orion such that valuable insights on total network power consumption can be obtained. The LUNA framework [EP04] takes message flows as input and then derives a power profile of the network fabric, capturing both the spatial variance across the network fabric and the temporal variance across application execution time.

GEMS [MSB⁺05] [Wis07] is a set of modules for Simics and it enables detailed simulation of CMPs. GEMS simulates a SPARC multiprocessor system and enables the simulation of commercial software such as database systems running on the Solaris OS. The GEMS Opal module provides a detailed, aggressive, out-of-order processor model. The GEMS Ruby module provides a detailed memory system simulator. GARNET [APJ08] is a detailed network model incorporated inside GEMS which enables system level performance and power modeling of the interconnection network. GEMS supports shared memory programming model.

2.3.3 Acceleration Techniques for Modular Simulators

Acyclic scheduling by Pérez et al. [PMT04] and process splitting by Naguib et al. [NG07] tackle SystemC engines by removing some useless process wake-ups. They work at the level of SystemC engine and don't modify the source code of simulators.

A technique by Penry et al. [PFH⁺06] works on parallelizing cycle-level modular multi-core simulators by utilizing the particular monotonic module scheduling properties of the Liberty engine. Parallelization in [PFH⁺06] is applied at level of phase (a concept that is similar to clock cycle).

Ezudheen et al. parallelize the OSCI SystemC engine to enable multithreaded simulation [PCC⁺09]. A set of runnable processes, i.e., a chunk, are allocated to a thread.

A technique by Savoie et al. [SSG02] extracts a logical sequence of SystemC processes and maps them to a thread. It handles program dependence graphs like a compiler front-end and transforms source code of simulators. A technique by Patel et al. [PS05] utilizes threads to accelerate simulations based on the synchronous data flow (SDF) model by utilizing the concurrency of the SDF model and the application of static scheduling.

SysCellC by Kaouane et al. [KHH08] uses IBM Cell processor's hardware to simulate a SystemC based system. But it modifies SystemC language elements and imposes limitations on processes. Similarly, SCGPSim by Nanjundappa et al. [NPJS10] uses GPUs to accelerate SystemC simulation by converting SystemC based source code into CUDA programs.

Distributed simulation techniques [Tra04][CCD⁺08][HBHT08] carry out simulations using multiple copies of SystemC engines located on machines connected via network.

2.4 Resource Management Techniques

Cores of multi-/many-core processors run not only individual sequential/parallel applications, but also support a diverse mix of applications [Bor07] [ABC⁺06]. Hence, they are considered "resources" that need to be carefully managed.

First, we survey resource management for multi-/many-core systems. For application-specific multi-/many-core systems, the design methodologies proposed solve resource management problems off-line, in a static manner. However, for general purpose multi-/many-core systems, configurations of systems resulting from multiple applications are too dynamic and too complex to be modeled off-line such that runtime/online techniques are necessary [Bor07] [ABC⁺06] [CM10]. As we are interested in the general-purpose NoC based manycore systems, only the runtime resource management techniques will be discussed.

On the other hand, the resource management problem has been studied in contexts like supercomputers, parallel and distributed systems. So, we review submesh based processor allocation schemes that have been applied in 2D-mesh connected parallel and distributed systems due to similarities between these systems and NoC based manycore chips with 2D-mesh topology.

2.4.1 Runtime Resource Management Techniques

An online resource allocation heuristic is proposed [MMB07] to execute several real-time, streaming media jobs simultaneously on a MPSoC which consists of up to 24 cores connected by an AEthereal NoC [GDR05]. A job annotated with resource budgets computed at compile-time can be independently started or stopped by the user. Resources that meet a job's required resource budgets are found online by using low-complexity algorithms. The technique allows 95% of resources to be allocated while handling a large number of job arrivals and departures.

In [NMA⁺05], a run-time resource management scheme is proposed to efficiently manage an NoC containing fine grain reconfigurable hardware tiles which are connected by an NoC and task migration is applied in to improve the system performance by reconfiguring these tiles.

The OS is proposed in [NMV⁺04] to optimize communication resource usage with the right NoC support. This OS-controlled mechanism allows the system to operate effectively in a dynamic manner. The model of a resource manager operating under OS control has become the foundation for the online resource management for NoCs. An adaptive routing strategy is also presented in [NMV⁺04] to ensure the quality-of-service requests of various applications.

Carvalho et al. propose dynamic task mapping scheme in NoC-based heterogeneous MP-SoCs, targeting the channel load minimization, a key cost function to optimize the NoC performance, for improving the performance [CCM07].

A technique is proposed in [CM07] [COM08] for run-time application mapping onto homogeneous embedded multicore NoCs with multiple voltage levels. The proposed technique consists of a region selection algorithm and a heuristic for run-time application mapping which minimizes the communication energy consumption, while still providing the required performance guarantees. It allows for new applications to be added to the system platform with minimal inter-processor communication overhead.

A run-time strategy is proposed in [CM08] [CM10] for allocating the application tasks to platform resources in homogeneous multicore NoCs. The user behavior information is incorporated in the resource allocation process which allows the system to better respond to real-time changes and adapt dynamically to user needs. Several algorithms are then proposed for solving the task allocation problem, while minimizing the communication energy consumption and network contention.

Tessellation manycore OS [LKB⁺09] [CBC⁺10] takes an aggressive approach of preemptive multi-tasking in resource management. Tessellation targets at the resource management challenges of emerging client devices, including the need for real-time and QoS guarantees. It is predicated on two central ideas: Space-Time Partitioning (STP) and Two-Level Scheduling. STP provides performance isolation and strong partitioning of resources among interacting software components, called Cells. Two-Level Scheduling separates global decisions about the allocation of resources to Cells from applicationspecific scheduling of resources within Cells.

2.4.2 Submesh Based Processor Allocation Schemes

Processor allocation is responsible for choosing a set of processors on which parallel jobs are executed on parallel and distributed systems. Since processor allocation is an NP-complete problem [LL05], allocation schemes are usually implemented by using heuristics.

Various submesh based processor allocation schemes have been proposed for traditional 2D mesh-connected multiprocessor systems and broadly they can be divided into two categories: contiguous and non-contiguous [Aba06] [BMOKAM07]. Within contiguous allocation schemes, jobs are allocated distinct contiguous submeshes of processors for the duration of their execution. Within non-contiguous schemes, a job can execute on multiple disjoint smaller submeshes instead of waiting for a single submesh of the requested size and suitable shape. Both contiguous and non-contiguous schemes aim to maximize the utilization of processors and non-occupied processors can participate in allocation. Discussions of these schemes focus on issues such as fragmentation, system performance and algorithmic complexity.

Some of the well-known contiguous submesh allocation schemes are the two-dimensional buddy system [LC91], the first-fit (FF) and best-fit (BF) scheme [Zhu92], the adaptive scan scheme [DB93], the frame sliding schme [CT94], the free list with compaction [LHLB95], the busy list with compaction scheme [SP96], the best-fit allocation and a virtual submesh allocation scheme [KY98], the right of busy submeshes with rotation scheme [CC99] and the free-list submesh allocation scheme [Aba06] which is of interest to researchers as it is better than other contiguous schemes in terms of scalability and algorithmic complexity.

The number of non-contiguous schemes proposed is less than the contiguous ones as the contiguity is possibly not utilized in processor allocation. Several non-contiguous schemes such as random allocation, paging and multiple buddy are proposed in [LWLN97]. A non-contiguous

allocation scheme, referred to as greedy-available-busy-list, which can decrease the communication overhead among processors allocated to a given job, is proposed in [BMOKAM07] [BMOKA07] where request partitioning in the scheme is based on the submeshes available for allocation aiming at reducing extra communication overhead.

2.5 Runtime Thermal Management

2.5.1 Thermal Problems

Thermal problems like thermal emergencies, localized hotspots and temperature gradients cause performance downgrade, timing errors, shortened lifetime of devices, even device damages to processors [Ska03] [SSS⁺04]. The deep sources for thermal problems are unavoidable variances in chips [Bor05]. Thus, thermal problems are among the important factors which designers have to confront when they design reliable systems with unreliable components [Bor05].

Thermal problems can be attacked at various phases of the design of electronic devices, including packaging and cooling design, circuit design and architecture design [SD06]. After devices are fabricated, runtime thermal management techniques can be utilized to complement the design-time cooling solutions where runtime chip power consumption and thermal profile are monitored and thermal-aware control techniques are engaged to eliminate thermal emergencies when they occur. The goals of thermal management are eliminating thermal emergencies and achieving good heat balance over chips. Maintaining good heat balance throughout chips reduces the chance of thermal crises [Bor05] and also improves their lifetime [Lu04].

With the advent of processors having multiple cores, the chip temperature profile results from thermal interactions among cores and the situation of temporal and spatial temperature variations for cores on the same die is further complicated [Bor05].

2.5.2 Runtime Thermal Management Techniques

In [Pow04], the thermal problems on CMPs with SMT cores are studied by managing power density at the level of thread. This scheme has two features. One is heat-and-run SMT thread assignment to increase processor-resource utilization before cooling becomes necessary by co-scheduling threads that use complimentary resources; the other is heat-and-run CMP thread migration to migrate threads away from overheated cores and assign them to free SMT contexts

on alternate cores, leveraging availability of SMT contexts on alternate CMP cores to maintain throughput while allowing overheated cores to cool.

In [Mer05] [WB06], a mechanism is presented for determining the energy characteristics of tasks by means of event monitoring counters. An energy-aware scheduling policy is also proposed to strive to assign tasks to CPUs in a way that avoids overheating individual CPUs.

In [DM06b], authors explore various thermal management techniques that exploit the distributed nature of multicore processors. Firstly, they classify various thermal management techniques in terms of core throttling policy, whether that policy is applied locally to a core or to the processor as a whole, and process migration policies. Distributed control-theoretic DVFS improves throughput among the evaluated options. Further, they also design a mechanism which has a PI-based core thermal controller and an outer control loop to decide process migrations.

Two heat reduction solutions are as follows. In [Nar05], authors present and evaluate three temperature-sensitive loop parallelization strategies for array-intensive applications executed on bus-connected chip multiprocessors in order to reduce the peak temperature.

In [Nar06], authors propose a compiler-based approach that balances the computational workload across the processors of an NoC based chip multiprocessor such that the chances of experiencing a thermal emergency at runtime are reduced. Their proposed compiler-directed approach makes use of ILP (integer linear programming) and operates in two phases. In the first phase, it determines the largest mesh area that can be occupied by the parallel computation without exceeding the performance degradation tolerance specified. In the second phase, it splits the workloads of select processors across multiple mesh nodes to further eliminate potential hotspots by balancing out the power density.

2.6 Conclusion

As discussed in Sections 2.1 and 2.2, NoC based manycore processors will become the mainstream in various domains in the future. We are particularly interested in NoC based homogeneous manycore processors with 2D mesh topology which are suitable for embedded domain. These processors are called “embedded manycore NoCs” for brevity in this thesis.

Architectural simulation has been adopted as the research methodology in this research. However, simulators investigated in Section 2.3.2 are not suitable for research on embedded manycore NoCs. Simulators such as NNSE, Noxim and NIRGAM don’t connect industrial level

cores and cannot support coupled simulations of CPU cores and NoC. Orion and LUNA are limited to power simulation. The combination of GEMS and GARNET does support coupled simulations of cores and NoCs. However, the aggressive out-of-order CPU models used in GEMS are not appropriate for manycore systems where smaller and simpler CPU cores are desired (Section 2.1.4). *Therefore, simulators suitable for embedded manycore NoCs have to be developed.* Following the simulation technology trends (Section 2.3.1), we prefer a modular infrastructure that is open-source and suitable for general-purpose CPU based platforms, based on which desired simulators will be developed.

As embedded manycore NoCs have abundant cores and uncores, the simulations of their activities are envisioned to be slow. Nowadays, multicore computers are prevalent for speedup through parallelism. Hence, we prefer the techniques which can leverage parallel processing for acceleration. In addition, we don't adopt techniques that modify the source code of simulators because architectural simulators are coded in a way to imitate activities of hardware systems at the architectural level and commonly don't allow source code to be transformed.

As a modular infrastructure has been considered, we have surveyed the acceleration techniques suitable for modular simulations in Section 2.3.3. However, the surveyed techniques have their respective limitations in terms of parallel processing and transformation of source code of simulators. Techniques in [PMT04] and [NG07] have single-threaded simulation engines and cannot benefit from parallel processing. Technique in [SSG02] transforms source code of simulators. SysCellC in [KHH08] modifies SystemC language elements and imposes limitations on processes. Similarly, SCGPSim in [NPJS10] converts source code of simulators into CUDA programs. The engines of distributed simulation techniques [Tra04] [CCD⁺08] [HBHT08] are single-threaded and cannot benefit from parallelism. Moreover, they modify source code of simulators. *Thus, systematic techniques that aim at accelerating architectural simulations on multicore platforms should be explored so as to exploit parallelism.*

Suitable simulation framework and acceleration techniques will enable us to explore research problems of embedded manycore NoCs. For embedded homogeneous manycore NoCs, runtime resource management techniques are indispensable for high performance (Section 2.4). Related proposals [CM07] [COM08] [CM08] [CM10] discussed in Section 2.4.1 have been shown to work on NoCs with medium core counts but their scalability for manycore systems with large core counts has not been discussed. Additionally, the interactions among applications

for resource management have not been explored. *Hence, efforts should be taken to explore novel runtime resource management techniques that consider the scalability and interactions among applications on embedded manycore NoCs.* For the scalability, the previous submesh allocation schemes discussed in Section 2.4 could be possibly leveraged. In addition, according to the “utilization wall” (Section 2.1.2), it is rational to leave some cores out of computation temporarily in order to achieve other optimization goals on manycore systems if necessary.

The thermal problems due to high power consumption are critical for the reliability of many-core processors. As surveyed in Section 2.5, suitable thermal management techniques for NoC based manycore processors are absent in the literature. Moreover, existing proposals for systems with small number of CPU cores are based on a reactive strategy. *Therefore, further research efforts should be taken to explore opportunities for better runtime thermal management on manycore processors, especially taking parallel techniques and a proactive strategy into consideration.*

Table 2.1 summarizes the aforementioned research challenges for NoC based manycore systems. In the ensuing chapters of this thesis, we will address these challenges.

Table 2.1: Research challenges and existing work

Research Challenge	Existing Work	Section
Simulation platform	The investigated simulators are not suitable for research on embedded manycore NoCs because 1) they cannot support coupled simulation of CPU cores and NoC or 2) they have CPU models inappropriate for manycore processors	Section 2.3.2
Acceleration technique	The acceleration techniques investigated have limitations in terms of parallel processing and modification of source code of simulators	Section 2.3.3
Resource management	The scalability of existing strategies has not been discussed. Runtime interactions among applications for resource management have not been studied	Section 2.4
Thermal management	Suitable thermal management techniques for NoC based manycore processors are absent in the literature	Section 2.5

Chapter 3

Simulation Framework for Network-on-Chip Based Manycore Systems

3.1 Introduction and Motivation

As discussed in the literature review, NoC-based manycore systems are considered as emerging platforms of significant importance and there are already numerous research activities aimed at these NoC based platforms.

Architectural simulators are widely used for evaluating different designs without building costly physical hardware and for obtaining detailed performance metrics. So, suitable simulators for NoC-based manycore systems are indispensable. However, little study has been reported on simulators targeting at NoC-based manycore systems, especially the simulators that integrate industry-level small and energy-efficient CPU cores into NoC and execute parallel applications. Here, parallel applications are those that support any of the popular parallel programming models like Open Multi-Processing (OpenMP), Message Passing Interface (MPI) and POSIX Threads (Pthreads). Hence, we are motivated to develop a simulation framework, which is used to construct desired architectural simulators, to fill the gap.

This chapter describes our work of developing a modular cycle-level micro-architectural simulation framework using the UNISIM environment [ACG⁺07, UNI09]. The simulation framework can be used to construct simulators supporting coupled simulations of NoC and small and energy-efficient cores for NoC-based manycore systems, which solves the simulation platform challenge as discussed in Section 2.6. In addition, we construct an example simulator for NoC-based manycore systems, which support execution of message-passing based parallel

applications, to show the extensibility of the simulation framework. Moreover, this simulator and related tools are used for other research issues presented in this dissertation.

We choose the UNISIM environment as the basis of our work because UNISIM is an open environment and it addresses several important issues of micro-architectural simulations in a unified fashion [ACG⁺07]. In order to exploit the benefits brought by UNISIM environment, much effort had been spent in understanding the source code of UNISIM environment before we got started to develop the simulation framework.

3.1.1 Contributions and Chapter Organization

The contributions of this chapter are as follows:

- a modular cycle-level architectural simulation framework has been developed to construct simulators that can support coupled simulations of NoC and CPU cores for NoC-based manycore systems;
- an example simulator for embedded manycore NoCs, together with related GNU based tool chain, has been developed to enable the exploration of research issues in the ensuing chapters;
- the simulation framework is open source and can be reused by other researchers because the developed modules of NoC can be easily changed or replaced by others, enabling fast customization and easy upgrading.

The rest of this chapter is organized as follows. Section 3.2 discusses the necessary background of UNISIM and NoC. Section 3.3 introduces a baseline architecture of NoC-based manycore chips for which the framework is built. Section 3.4 elaborates on the design and implementation of the framework. Section 3.5 describes how we test the framework before using it to construct simulators. Section 3.6 describes the details of building a simulator using the framework and the experimental results of simulating a parallel program on it.

3.2 Preliminaries

3.2.1 UNISIM Environment

Our simulation framework relies on the UNISIM environment. UNISIM is a structural simulator environment built on top of SystemC for hardware modelling. It was proposed and developed

to overcome the many shortcomings of existing simulation techniques. The five major features of UNISIM are as follows:

- (i) UNISIM is a modular simulation environment, implemented as a layer on top of the industry standard SystemC [OSC09]. Besides modularity, the UNISIM environment focuses on the reuse of control logic, which corresponds to a large share of simulator code, and which is often overlooked by simulation environments.
- (ii) UNISIM supports an abstract level of modeling, called Transaction-Level Modeling (TLM), in addition to the more common detailed Cycle-Level Modeling (CLM). TLM simulators are less accurate but much faster than CLM simulators. UNISIM allows hybrid CLM/TLM simulators which can zoom in on only the important architecture details.
- (iii) UNISIM contains full-system functional simulators capable of booting a complex operating system like Linux. These functional simulators can be plugged into CLM or TLM simulators which are compliant to a functional simulator API.
- (iv) UNISIM provides APIs for a set of services. Any module implementing this set of standardized calls automatically benefits from the corresponding services. Moreover, since these services are provided at the simulation engine level, i.e., independently of any simulator, they can be easily modified or replaced.
- (v) UNISIM provides a library of compatible modules and models which comes with the environment and is open to external contributions. UNISIM also has the ability to interoperate with other simulators by wrapping them into UNISIM modules.

As mentioned above, because cycle-level simulators built upon UNISIM environment have high accuracy with respect to performance evaluation, our framework is developed as a cycle-level model.

The kernel component of UNISIM cycle-level modeling consists of a compilation engine and a library. The library defines modules, ports, signals, etc. The compilation engine produces C++ source files of a simulator from the set of leaf and hierarchical modules defining it. The produced files are compiled and linked to the library to create the simulator executable.

There is an online repository of the UNISIM environment [UNI09]. It has many cycle-level modules for reusable hardware blocks and several cycle-level simulators for case study.

UNISIM has been successfully used to develop complex cycle-level simulators including Cell-Sim [Cel09] which models IBM's Cell, a modern heterogeneous multiprocessor.

3.2.2 Cycle-level Modeling in UNISIM

UNISIM provides a set of rules for modular hardware modeling. A hardware module is a derived class of UNISIM class *module* and can have states. A module only exposes interfaces and its implementation is hidden.

A hardware module can be reused in different designs as a plug-in. Modules can be connected through ports. A connection is established when an output port is linked to an input port of the same type through signals. A connection has three different signals: *data*, *accept* and *enable*. The signal *data* is typed signal to pass data of various types. The signals *accept* and *enable* are simple boolean signals. By establishing connections, a set of modules could define a hierarchical hardware architecture.

UNISIM also provides a well defined communication protocol to implement the interactions between pairs of modules. Passing values between two modules is as follows: a sender module writes data to its output ports and a receiver module reads data from corresponding input ports. Processes are designed in modules to define their behaviors in response to changes in signals of ports. A communication transaction is initialized at a sender module by sending data via *data* signal of output ports; the receiver module can accept them or not by setting the *accept* signal of the corresponding input port to **true** or **false**; the sender can enable or disable the transaction by setting the *enable* signal of the output port to **true** or **false**. In this way, centralized controls are distributed among modules with connection signals in UNISIM-based simulators.

UNISIM designs service/client relations via interfaces to enable interactions without using hardware ports. An interface is a set of standardized calls which acts as a contract between service and client. Any module importing a service automatically benefits from it. UNISIM utilizes services to implement many simulator-independent functionalities which are technology- and software- related. Services are easy to modify or replace due to their independency. Several examples of services are: loader, thread, memory, debugger, OS and syscall etc. All registered services are managed by a Service Manager which helps interaction and debugging. Developers can give names to modules and ports for debugging purpose. Moreover, APIs are provided to check signals during simulation to avoid the "unknown" status.

3.2.3 Network-on-Chip

3.2.3.1 Topology

The topology of a network is the specific way in which the nodes and links are connected. For on-chip networks, the topology determines the physical layout and connections between nodes and channels in the network. While an on-chip network may be of arbitrary topology, commonly they have a regular structure. Figure 3.1 shows several possible topologies of on-chip networks.

Figure 3.1: Several topologies of Network-on-Chip

Topology of NoC has profound effects on the overall cost and performance of the on-chip network. First, the topology determines the number of hops (or routers) a message must traverse and the physical interconnect lengths between hops. Thus, network latency is significantly influenced by the topology. Second, as energy is incurred when messages traverse via routers and links, the hop count decided by the topology also directly affects network energy consumption. Third, the topology dictates the total number of alternate paths between nodes which affects how well the network can spread out traffic and support bandwidth requirements.

Several metrics have been commonly used in comparing topologies at the early stages of design [PJ09]. They are listed as follows:

- *Degree.* The degree of a topology refers to the number of links at each node. It can be used as an abstract metric of a network's cost, as a higher degree requires more ports at routers, which increases implementation complexity.

- *Hop count.* The hop count is defined as the number of hops a message takes from source to destination, or the number of links it traverses. It is a very simple and useful proxy for network latency.
- *Maximum channel load.* It is defined as being relative to the injection bandwidth. *Injection bandwidth* is defined as the bandwidth between a node's network interface and a network router. When the load on a channel is said to be 2, it means that the channel is loaded with twice the injection bandwidth. It is useful as a proxy for estimating the maximum bandwidth the network can support, or the maximum number of bits per second (bps) that can be injected by every node into the network before it saturates.
- *Path diversity.* It is defined as the number of multiple shortest paths between a given source and destination pair. Path diversity within the topology gives the routing algorithm more flexibility to load-balance traffic and to route around faults in the network.

In addition, the implementation complexity cost of a topology depends on the following factors: 1) the degree of nodes and 2) the ease of laying out a topology on a chip, which are decided by wire lengths and the number of metal layers required.

3.2.3.2 Routing

The topology of the network defines its physical structure, but the routing algorithm is what defines what paths of links and routers individual packets take when travelling from their sources to their destinations. The routing algorithm is used to distribute traffic evenly among the paths supplied by the topology, so as to avoid hotspots and minimize contention, thus improving network latency and throughput. All performance goals must be achieved under tight constraints on implementation complexity: routing circuitry can stretch critical path delay and add to a router's area footprint. While the energy overhead of routing circuitry is typically low, the specific route chosen affects hop count directly, and thus substantially affects energy consumption.

Generally, routing algorithms can be divided into three classes: *deterministic*, *oblivious* and *adaptive*. Under a *deterministic* routing algorithm, a packet will always take the same path between the same nodes. While numerous deterministic routing algorithms have been proposed, the most commonly used routing algorithm in on-chip networks is *dimension-ordered routing (DOR)* due to its simplicity. With DOR, a message traverses the network dimension

by dimension, reaching the coordinate matching its destination before switching to the next dimension.

The second class of routing algorithms are *oblivious* ones, where messages traverse different paths, which are oblivious of routing from source to destination, but the paths are selected without regard to network congestion. In this way, these routing algorithms can be kept simple. Deterministic routing can be considered as a subset of oblivious routing.

The third class of routing algorithms can be *adaptive*, in which the path a message takes from source to destination depends on network traffic situation, such as buffer capacities, energy constraints, link load, packet size or content, or network faults.

Another way in which routing algorithms can be categorized is whether they are minimal or not. *Minimal routing algorithms* select only paths that require the smallest number of hops between the source and the destination. *Non-minimal routing algorithms* allow paths to be selected that may increase the number of hops between the source and destination. The non-minimal routing algorithms have the advantage of avoiding network congestion. In the absence of congestion, non-minimal routing increases latency and also power consumption as additional routers and links are traversed by a message. With congestion, the selection of a non-minimal route that avoids congested links, could result in lower latency for packets.

Of chief concern for routing algorithms is that they are either deadlock-free or can recover from deadlock. A deadlock occurs when a cycle exists among the paths of multiple messages. In detail, deadlock is the condition in which two or more packets are waiting on network resources (for example, input buffers) to be freed and simultaneously holding those resources which the other packets are requesting, in a circular fashion. As a result of deadlock, no progress is made.

Most complex dynamic routing algorithms often require either comparatively complex deadlock recovery schemes or more complex microarchitecture with which deadlock can be avoided in the routing algorithm by preventing cycles among the routes by the algorithm, or in the flow control protocol by preventing buffers from being acquired and held in a cyclic manner.

The principle benefit of dimension order routing is its simplicity. Moreover, it is easier to design in a deadlock-free manner. However, it eliminates path diversity in a mesh network and thus lowers throughput as only one path exists between source and destination with this routing. Without path diversity, the routing algorithm is unable to route around faults in the network or avoid areas of congestion. Therefore, this routing does a poor job of load balancing the network.

3.2.3.3 Flow Control

Flow control decides the allocation of network buffers and links. It determines when buffers of routers and links are assigned to messages, the granularity at which they are allocated: granularity at which they operate: either at the flit, packet-level, or message-level, and how these resources are shared among the many messages using the network.

A well-designed flow control protocol lowers the latency experienced by messages at low loads by not imposing high overhead in resource allocation, and maximizes the amount of resource sharing so that message-blocking is minimized and therefore less storage is required for high network throughput. In determining the rate at which packets access buffers and traverse links, flow control is instrumental in determining network energy and power consumption.

Flow control can be message-based, packet-based and flit-based, different at the resource allocation granularity. The technique used for message-based flow control is called *circuit-switching* [DYL02] [DT03]. Circuit switching pre-allocates resources (links) across multiple hops to the entire message. With packet-based flow control techniques, messages are broken down into multiple packets and each packet is handled independently by the network. *Store-and-forward* [DYL02] [DT03] flow control is an example for that. To reduce the buffering requirements of packet-based techniques, flit-based flow control mechanisms are introduced. One good example of flit-based flow control is wormhole flow control [Dal90] [Dal92] which is commonly adopted in on-chip interconnection networks.

As low buffering requirements help routers meet tight area or power constraints for on-chip networks, many on-chip networks adopt flit-based flow control. The NoC of Intel's Polaris [HVS⁺07] uses flit-based wormhole flow control with two virtual channels, though the virtual channels are used only to avoid system-level deadlock, and not for flow control. This simplifies the router design since no VC allocation needs to be done at each hop. For the Tile processors of Tileria [WGH⁺07], the iMesh's four dynamic networks use simple wormhole flow control without virtual channels to lower the complexity of the routers, trading off the lower bandwidth of wormhole flow control by spreading traffic over multiple networks.

3.2.3.4 Router Micro-architecture

The microarchitecture of a router is in large part determined by the network's topology, routing algorithms and flow control. In turn, the microarchitecture determines the power and performance characteristics of the NoC. A router's architecture determines its critical path delay which

affects performance including per-hop delay and overall network latency. Router microarchitecture also impacts network energy as it determines the circuit components in a router and their activity. The realization of the routing, flow control and the router pipeline will affect the efficiency at which buffers and links are used and thus overall network throughput.

A typical router's microarchitecture consists of input and output channels, routing logic, virtual channel allocation logic, switch allocator, a centralized crossbar, and input buffers. The virtual allocation logic is where the majority of the flow control functionality is implemented. The crossbar is fully connected so that any set of input-output port combinations can be satisfied in one cycle as long as no two input ports are equal and no two output ports are equal. In other words, as many flits up to the degree of the crossbar may traverse the crossbar in one cycle as long as no two flits originate from the same port and no two flits are destined for the same port. Buffers are used to house packets or flits when they cannot be forwarded right away onto output links. Flits can be buffered on the input ports and on the output ports. Output buffering occurs when the allocation rate of the switch is greater than the rate of the channel. The above router is input-buffered, in which packets are stored in buffers only at the input channels because input buffering permits the use of single-ported memories. The organization of buffers has a large impact on network throughput, as it heavily influences how efficiently packets share link bandwidth. Input and output channels connect the router to the neighbouring routers or a local network interface which itself is connected to the local processing element.

The processing of flits within a virtual channel router can be considered as several logical pipeline stages. A head flit, upon arriving at an input port, is first decoded and buffered according to its input virtual channel in the *buffer write (BW)* stage. In the next stage, the routing logic performs *route computation (RC)* to determine the output channel for the packet. The header then arbitrates for a virtual channel corresponding to its output channel in the *virtual channel allocation (VCA)* stage. Upon successful allocation of a virtual channel, the header flit proceeds to the *switch allocation (SA)* stage where it arbitrates for the crossbar switch input and output ports. After winning the output port, the flit is then read from the buffer and proceeds to the *switch traversal (ST)* stage, where it traverses the crossbar. Finally, the flit is passed to the downstream router in the *link traversal (LT)* stage. Body and tail flits follow a similar pipeline except that they do not go through RC and VCA stages, instead using the route and the virtual channel allocated by the head flit. The tail flit, on leaving the router, deallocates the virtual

channel reserved by the head flit. If a wormhole router does not support virtual channels, the virtual channel allocation stage can be omitted.

We can see the major features of the routers in some example implementations. The routers of Intel's Polaris [HVS⁺07] are aggressively pipelined. Each router uses a 5-stage pipeline: buffer write, route computation, two separable stages of switch allocation, and switch traversal. Each input channel has two input queues of buffers, one for each virtual channel, that are each 16 flits deep. The switch allocator is separable in order to be pipelineable. The virtual channels are not leveraged for bandwidth, but serve only deadlock avoidance purposes. The multiple on-chip networks of Tile processors by Tiler [WGH⁺07] have a single-stage router pipeline during straight portions of the route, and an additional route calculation stage when turning. Only a single buffer queue is needed at each of the 5 router ports, since virtual channels are not supported. Only 3 flit buffers are used for each input channel, just sufficient to cover the buffer turnaround time. Simple routers are designed for a low area overhead.

3.2.3.5 Network Interface

The Network Interface (NI) is considered as the glue logic necessary to connect the components inside a tile such as compute cores to the router of NoC. Services provided by NI can be classified into the following categories: core adaptation, clock adaptation, network and functional [BM06a].

Core adaptation services including core interfacing and packetization are the basic core wrapping services and their role is to adapt the communication protocol of the component to the communication protocol of the network. Core interfacing provides a high performance physical connection between the NI and the corresponding core with a layering concept applied. The packetization service takes the incoming signals specifying processor core transactions and builds packets respective to the NoC communication protocol.

The clock adaptation service is necessary because SoCs or processors will probably be designed in the style of locally synchronous and globally asynchronous (GALS). Even if the clock frequency is the same over the chip, phase adaptation is needed for communications. On-chip networks are composed of simple elements and rely on path segmentation, and thus can run at higher frequencies to decrease the latency seen by the computation units as a separate design and optimization of the data-path and of the control path can potentially lead to higher clock speed implementations.

Network services include transactions ordering, reliable transactions and flow control. Transaction ordering is needed in networks where dynamic routing schemes are implemented as packets can potentially arrive unordered and this is typically not acceptable because it raises a memory consistency issue. The NI must reorder the transactions before forwarding them to the core. On-chip communication medium is considered not reliable in future deep sub-micron technologies. The NI could be involved in providing reliable network transactions by inserting parity check bits in packet tails or by implementing end-to-end error control. Flow control is in-charge of regulating the flow of packets through the network, and of dealing with localized congestion. The NI is involved with the generation of flow control signals exchanged with the attached router. The NI can also provide packet flow control given that the necessary amount of decoupling buffering resources is implemented in the NI. This allows the system to decouple (to a certain extent) core computation from its requests for non-blocking communication services.

Functional services add new functionalities to the system. Several examples of them are cache coherence, security and low power. Cache coherence on an NoC is no longer an easy task because snooping is difficult. New protocols are needed to allow the use of NoC based multi-/many-core systems at low cost. In sensitive SoCs, the security of transactions is important. The NI could offer to cores an encryption service in order to prevent Tempest-like pirating (electromagnetic emanation analysis of the chip). In addition, the NI can be used to filter the target addresses and allow only some communication, which is useful to prevent a sensitive part of the chip from communicating with another part. The power concern becomes even more critical in NoC based systems. Beyond low level low-power design techniques, higher-level techniques are likely to achieve larger savings. For instance, switching off some components and waking them up is a system-level power management technique that could be applied to NoC building blocks (especially NI).

A typical NI architecture contains front-end and back-end sub-modules. The NI front-end implements a standardized point-to-point protocol allowing core reuse across several platforms which allows core developers to focus on developing core functions without the advance knowledge regarding potential end-systems. The NI back-end provides the services of the network layer, the link layer and the physical layer. Data packetization and routing-related functions can be viewed as essential tasks performed by the network layer, and are tightly interrelated. The NI back-end also provides data-link layer services such as communication reliability and

flow control. The physical channel interface to the network router has to be properly designed to handle challenges such as clock-domain crossing, high-frequency link operation, low-swing signalling, and noise-tolerant communication schemes.

3.3 Baseline Architecture for NoC-based Manycore Systems

In this section, a preliminary and extensible architecture for NoC based multi-/many-core chips is described. The chosen techniques of topology, routing, flow control, router microarchitecture, network interface are based on discussions in the Preliminaries section. The modular design of this simulation framework allows for easy updates and modifications.

Though different topologies can be implemented by changing the connections between routers, the chosen topology of its on-chip network is mesh because mesh is regular, simple and predictably scalable with regard to power and area [BD06]. The routers are implemented as pipelined and support wormhole switching and virtual-channel flow control. Various parameters of on-chip network, such as dimension, sizes of flit and packet, routing algorithms, etc., are configurable. Various cores can be connected to NoC via network interfaces and network access operations are explicitly exposed to cores. This architecture can serve as a foundation for building NoC based multi-/many-core chips with different and sophisticated architectures.

3.3.1 A Network-on-Chip and Its General Structure of Tiles

Figure 3.2.a shows an NoC with its tiles arranged as 4×4 2D mesh. Each tile has its coordinates and a unique identity (*id*). The coordinates of the tiles at the left-top corner and the right-bottom corner are $(0, 0)$ and $(3, 3)$ respectively. A tile's *id* is calculated with its coordinates (x, y) following the formula:

$$id = y * width_of_mesh + x.$$

The above NoC implements techniques such as wormhole switching and virtual-channel flow control. Here, a credit-based flow control mechanism [BM06a] is adopted. Each tile contains a router and routers are connected to ones in neighboring tiles with physical links.

In an NoC supporting switching technique, the granularity of data transfer is usually defined as follows, as discussed in the literature review. The unit of data transferred in a single cycle on a link is called a *phit* (physical unit). The unit of data for synchronization at link-level flow

 3.2.a: An NoC as 4×4 mesh

3.2.b: General structure of tiles

Figure 3.2: Structures of an NoC and its tiles

control is called a *flit* (flow control unit) and a flit is at least as large as a phit. Multiple flits constitute a *packet*. Further, several packets make up *messages* that modules connected to NoC send to each other.

Different NoCs can use different sizes for phit, flit, packet and message. *Æthereal* [GDR05] uses phits of 32 bits, flits of 3 phits, and packets and messages of unbounded length. *SPIN* [GG00] uses phits and flits of 36 bits, and packets can be unbounded in length. For simplicity, in the NoC of our framework, a flit has the same size of a phit. Moreover, the size of a flit is configurable for users.

In the NoC of our framework, the first flit of a packet, i.e., header flit, contains routing information, such as source and destination tile ids, which is used by routers to decide header flit's route. Other flits of the same packet, i.e., body and tail flits, follow header flit's route. A flit may traverse several intermediate routers until it arrives at the destination tile.

The general structure of tiles in above architecture is shown in Figure 3.2.b. A tile contains one or more cores/uncores, other hardware components, a network interface (NI) and a router. A router is regarded as a part of a tile because a router is commonly put inside a tile on the physical floorplans on many NoC-based chips. An NI acts as a bridge between a router and resources in the same tile.

Figure 3.3: A typical virtual channel router

3.3.2 Virtual Channel Router

A router is the primary component of NoC. An example of a virtual channel router for 2D mesh topology, which is similar to those in [PD01] [MWM04] [HVS⁺07], is shown in Figure 3.3. Its major components are: several pairs of physical input/output channels, virtual channels (including buffers) for physical input channels, routing logic, a virtual channel allocator, a switch allocator, and a crossbar switch. Credit input/output lines exist between routers which pass information of availability and buffer status of virtual channels.

The numbers of pairs of physical channels are variable for routers at different locations of mesh. For the routers located at corners, the number of pairs is 3. One pair of input/output channels are connected from/to the NI in the same tile. Other two pairs of input/output channels are connected from/to routers of neighboring tiles. For routers on the borders of and internal to the mesh, the numbers of pairs are 4 and 5 respectively.

3.4 Design and Implementation of the Framework

This section first describes the layers of design of the framework. Then, design and implementation of the most important module, virtual channel router, and its inner modules are elaborated. Further, the considerations in NI's design are discussed. Finally, the configurable parameters and the performance metrics of the framework are introduced.

The simulation framework was developed on Linux and compiled with GCC 4.2.3. UNISIM environment is kept at the version of May 17, 2008.

3.4.1 Layers of Software Design

Following the practices of UNISIM, the framework is designed and implemented in layers. At the innermost layer, each tile manages its inner modules such as router, network interface, compute core and other components. The number of ports in a router is dynamically managed according to its tile's location in the mesh.

The middle layer of the simulator manages all tiles and connections between them. At this layer, the clocks of all tiles are set to the global clock. The connections between input and output ports of tiles are established here. Users can change the topology of the network-on-chip simulator by modifying connections between tiles.

The outermost layer implements functionalities such as instantiation of a simulator and control of execution. During the instantiation of the simulator, various modules are created and connected. Additionally, services provided by UNISIM are used to setup executable images of cores and to complete initialization of memory. We also adopt the good practice that all signals of the simulator are checked at every cycle. The command line is parsed to get input parameters from users and these parameters are used to configure the simulator and to control its behaviors.

3.4.2 Virtual Channel Router and Its Inner Modules

The virtual channel router in Figure 3.3 is designed as a UNISIM hardware module which contains other components. Some of these components are designed as hardware modules: input channel, output channel, virtual channel allocator and switch allocator. Remaining components, routing logic and crossbar switch, are common C++ classes.

The variable numbers of ports/connections of routers are calculated according to the coordinates of tiles. Subsequently, pairs of input and output channel modules are created when a router is instantiated.

3.4.2.1 Routing Logic (RL) and Crossbar Switch (CS)

Routing logic is implemented in a common C++ class named *RoutingAlgorithms*, instead of a hardware module. Resource request conflicts are handled at a module's virtual channel allocator and switch allocator. Routing algorithms are designed as static methods of this class. Each method accepts a routing request and returns a routing result. A routing request contains

information such as the *id* of the current tile, and the *ids* of source tile and destination tile. A routing result indicates the direction of the physical output channel.

Currently implemented routing algorithms include X-Y routing (a simple deterministic routing [PGJ⁺05]), Odd-Even adaptive routing [Chi00], Fully Adaptive routing [ZL01] and table-based routing. The table-based routing is implemented by adding to each router a routing table whose entries define routing information between source and destination nodes. Data of tables are loaded from configuration files when routers are instantiated. Other routing algorithms, such as source routing, DyAD adaptive routing [HM04] and NoP adaptive routing [ACPP08], can be added by introducing relevant data structures and methods.

The crossbar switch is modeled by connecting each output port of input channels to each input port of output channels. The connections are setup when the router is instantiated. Control of this switch is placed in switch allocator module. When an input channel is granted with permission of passage, it can send out flits. Otherwise, it has to wait.

3.4.2.2 Input Channel (IC)

Input channel module manages multiple virtual channels (VCs) and flit buffer slots for them. The numbers of VCs and buffer slots of each VC are configurable. Buffer of a VC are organized as a FIFO queue. Each VC stores the routing result of handling a header flit which is applied to body and tail flits of the same packet.

A pipelined router processes flits in steps. In our design, an input channel module contains five inner modules to process flits: *stage buffer*, *stage route compute*, *stage VC allocation*, *stage port allocation* and *stage switch traversal*. A pipeline with six stages formed by these modules is discussed later.

3.4.2.3 Virtual Channel Allocator (VCA)

When a header flit is processed in an IC, *stage VC allocation* module sends a request to the VCA to acquire a virtual channel id (*vcid*) in next router or local NI. More than one such requests in the same cycle lead to conflicts and VCA arbitrates these requests following service policies.

In addition, the VCA keeps values of credits of neighboring routers and the local NI. These values are updated based on inputs from credit input lines from neighboring routers and local NI. They are used together with service policies in making allocation decisions.

3.4.2.4 Switch Port Allocator (SPA)

After flits obtain the route information including $vcid$ and output physical channel, they come to *stage port allocation* modules of ICs. These modules send requests to switch port allocator to acquire permissions for flits to pass the crossbar switch. Similar to the VCA, in the same clock cycle, there could be conflicts due to multiple requests. Relevant service policies have to be applied to solve these conflicts.

3.4.2.5 Output Channel (OC)

Major functionalities of OC modules include receiving flits passed from output ports of ICs and sending these flits to ICs of next routers or local NI. Accordingly, each OC module has two categories of registers to store flits for buffering. One category is for storing the flits passed from ICs and there is one register for each IC. These registers are named *input registers*. The other category is for storing the flits which will be sent to VCs of IC in next router or local NI. The number of registers of this category equals to the number of VCs of IC in next router or local NI. These registers are named *output registers*. OC also moves flits from the input registers to appropriate output registers.

3.4.2.6 Pipeline of Router

Figure 3.4: Pipeline of the virtual channel router

The traversal of flits of a packet through the above router can be divided into several steps which form a pipeline with six stages shown in Figure 3.4. These stages are described as follows:

1) (Buffer writing) When a header flit h of a packet p arrives at a router r from one of its ICs, $vcid$ contained in h is read and flit h is stored into a buffer slot of the virtual channel VC_{vcid} if VC_{vcid} is not occupied. After h is saved, VC_{vcid} is marked as “occupied” and the number of available buffer slots of VC_{vcid} is reduced by 1. VC_{vcid} is occupied by packet p

until all p 's flits are transferred. Credit values of this IC which reflect status of resources are passed to the neighboring routers via “credits out” lines. This stage is called “buffer writing” and implemented in the *stage buffer* module.

2) (Route compute) Then, flit h is passed from *stage buffer* module to *stage route compute* module. Here, a request for routing for h is sent to routing logic. A reply contains routing result, i.e., the OC to be used by h , is received and saved in related data structure of VC_{vcid} . This stage is called “route compute”. After this stage, h is sent to *stage VC allocation* module for virtual channel allocation.

3) (Virtual channel allocation) A request for allocating a VC in next router or local NI is sent to VCA for flit h . VCA makes decision based on the targeted OC, current service policy (such as round-robin) and the availability of VCs in next router or local NI (passed from “credits in” lines). A reply containing the allocated virtual channel id, denoted as $nextvcid$, is sent back from VCA if $VC_{nextvcid}$ is available. If no VC is available, the reply contains a specified value, i.e., $VCID_NULL$, and requests for allocation keep being sent in following clock cycles until a VC is allocated to h . The $nextvcid$ is also saved at VC_{vcid} and $vcid$ in the header flit h is updated to $nextvcid$. The routing information for h saved at VC_{vcid} will be used by the body and tail flits of the same packet p for their transfers. This stage is named “virtual channel allocation”.

4) (Switch allocation) Next, flit h is passed to *stage port allocation* module. For each flit, including header, body and tail, a request for switch passage permission is sent to SPA. SPA makes port allocation decision based on related service policies. SPA controls the crossbar switch by granting permissions to corresponding ICs. This stage is named “switch allocation”.

5) (Switch traversal) Flits granted passage permission for the crossbar are passed to input ports of the appropriate OCs. When the tail flit of packet p passes the crossbar switch, the VC of the current IC which has been allocated to p can be freed for transfers of other packets. This stage is named “switch traversal” and implemented in the *stage switch traversal* module.

6) (Link traversal) The flits at the OCs are transferred to next routers or local NI. This stage can be regarded as implemented in output channel module. After this, flits are transferred to a neighboring router or local NI.

In Figure 3.4, there is a bypass path between stage *buffer writing* and stage *switch allocation*. It is used by the body and tail flits which don't need to go through the stages such as “route compute” and “virtual channel allocation”.

3.4.3 Network Interface (NI)

NIs are usually considered as the glue logic necessary to interface compute cores to the NoC. A typical NI in physical implementation contains front-end and back-end submodules [BM06a] [RDP⁺05]. Functionalities of an NI could include core interfacing, packetization, flow control, clock adaptation, reliability, security, etc. Since we are implementing NI for architectural simulators, the functionalities of NI in our framework are simplified to only include core interfacing, packetization and flow control. Moreover, the cores, NIs and NoC share a common clock.

Core interfacing and packetization are closely related. The communication between cores, i.e., message passing between cores, comprises of three stages: the packet assembly, packet transmission and the packet disassembly and delivery.

The commonly used packetization strategies are software library based, on-core module based and wrapper based [BM03]. In our framework, the on-core module based strategy is chosen since it has low latency and high flexibility at modest cost of hardware complexity including additional registers and logic and an increase in instruction set [BM03]. In addition, this strategy makes network access operations directly exposed to the ISA which paves the way for a tighter coupling of computation and communication [BM06a].

The flow control of NI involves interactions with both router and compute core connected to it [BM06a]. When the NoC cannot accept new packets any more because of congestion, NI can still accept new transactions from connected core if it has enough decoupling buffering resources. To a certain extent, this mechanism can decouple core computation with its requests for non-blocking communication services. However, when NI runs out of buffers, the congestion in NoC has impact on the core's behavior in that the core has to be stalled if it requires additional communication services.

3.4.4 Configurable Parameters of the Framework

Several parameters of the framework are designed as configurable. Some can be set when simulators are compiled and others can be changed when simulators run. These parameters and their brief descriptions are as follows.

- *NoC size* is specified with number of rows and number of columns of NoC.
- *Flit size* is the maximum number of bytes one flit can accommodate.

- *Packet size* is the maximum number of flits a packet can accommodate.
- *Routing algorithm* is used by routing logic in routers. Currently supported routing algorithms in the framework are: XY, Odd-Even, Fully Adaptive, Table-based.
- *Routing table file* is a text file containing information for routing tables of routers.
- *VC number* is the number of virtual channels each physical channel supports.
- *Buffer depth of virtual channel* is the number of buffers for flits at each virtual channel. A buffer slot accommodates a flit.
- *Buffer depth of NI* is the size of buffer (in packets) for messages at each NI.
- *Service policy* is the policy used to arbitrate conflicts at virtual channel allocator or switch port allocator. The default policy is first-come-first-served (FCFS).

3.4.5 Metrics for Performance Evaluation

Similar to [PGJ⁺05][ACPP08], two metrics, *throughput* and *average delay*, are chosen in our framework to evaluate performance of the NoC.

For message-passing, metric *throughput* (TP) is defined as follows:

$$TP = \frac{\text{Total received flits}}{\text{Number of nodes} \times \text{Total cycles}}$$

where *Total received flits* refers to the number of whole flits that arrive at their destination nodes, *Number of nodes* is the number of network nodes, and *Total cycles* is the number of clock cycles elapsed between the occurrence of the first message generation and the last message reception. Thus, message throughput is measured in flits/cycle/node. It is the fraction of the maximum load ($TP = 1$) that the network is capable of physically handling, assuming that each node receives a flit in each cycle.

The delay of a message m is defined as the time in clock cycles that elapses between the occurrence of m 's header flit injection into the network at the source node and the occurrence of m 's tail flit reception at the destination node. Metric *average delay* (D) is defined as follows:

$$D = \frac{1}{N} \sum_{i=1}^N D_i$$

where N is the total number of messages reaching their destination nodes and D_i is the delay of message i .

This framework is different from other simulators in that it integrates compute cores to run parallel applications. Therefore, performance evaluation of NI is mandatory and *average overhead* is proposed for this. During the transfer of a message from source to destination, there are overheads in the source and destination NIs. These overheads result from the activities in NIs such as data movement, packetization and flow control etc. The concept of *lifetime* is introduced. The lifetime of a message m is defined as time in clock cycles that elapses between the beginning of m 's transferal at the source node and the occurrence of m 's consumption at the destination node. Thus, overhead of m equals to difference of its lifetime and its delay. Metric *average overhead* (H) is defined as:

$$H = \frac{1}{K} \sum_{i=1}^K H_i = \frac{1}{K} \sum_{i=1}^K (L_i - D_i)$$

where K is the total number of messages consumed by their destination nodes, D_i is the delay of message i and L_i is the lifetime of message i .

To calculate values of these performance metrics for simulations, detailed information is recorded in data structures of relevant components such as routers and NIs.

With above definitions of basic metrics, several derived metrics could also be defined. Several examples of them are: global average throughput, local throughput, global average delay, maximum/minimum global delay, global/local average overhead, etc.

3.5 Testing the Simulation Framework

The simulation framework is utilized to construct a simulator which is similar to Noxim [Nox08] and NIRGAM [Nir07]. This simulator doesn't consider particularities of applications. Instead, it is used to test and debug the simulation framework on routing algorithms, flow control, router microarchitecture and so on.

In this simulator, simple processing elements (PEs) are embedded in tiles. Some of PEs generate and inject packets into the on-chip network, which are called "sending element/sender", while the others accept packets from the on-chip network, which are called "receiving element/receiver". For this simulator, foci are put on the performance metrics, routing algorithms,

Figure 3.5: A simulator for testing the developed simulation framework

router microarchitecture, etc. Though we still use mesh as the topology of simulator, the topology of NoC can be changed to other topologies as described in Section 3.4.1.

Shown in Figure 3.5, a simple processing element is embedded in each tile. The processing element communicates with processing elements in other tiles through the on-chip network which consists of network interfaces and routers. As our purpose is to test the framework, we only consider a half-duplex system for simplicity. Some processing elements can act as senders, which send out packets/flits as source nodes, while some other processing elements can act as receivers, which receive packets/flits as sink nodes. As shown in Figure 3.5, two example processing elements, a sender and a receiver, have been zoomed in.

The network interface is connected to the processing element in the same tile through an input port and an output port. When a sender processing element needs to send a packet, it sends to the network interface a command which contains information of the packet: the id of packet, the length of payload in bytes and destination processing element. As each command can contain data of fixed length, data are divided into parts which are treated as payloads of packets such that each packet can be accommodated by a command.

Each network interface has two finite state machines: one for sending packets and the other for receiving packets. When a network interface receives a command from a sender processing element, it enters the sending state machine to send the packet. The packet could be sent out to the connected router in several flits based on length of payload. When flits of the packet arrive at the router in the tile of the destination processing element, the router forwards the flits to the network interface connected to the destination processing element. This network interface in

the destination tile enters the receiving state machine and restores the packet when all flits of it arrive. When the packet is ready, the network interface sends a command containing the packet to the processing element. The credit-based end-to-end flow control is implemented between network interfaces.

In addition to the simple traffic, different transport traffic can be realized by programming the sender processing elements which are briefly introduced as follows.

- It is easy to implement constant bit rate (CBR), i.e., transport traffic at a consistent bit rate between the traffic source and destination. Developers need to specify the packet size (in bytes), load percentage (percentage of channel bandwidth to be used), destination (a fixed destination or randomly chosen destination) and inter-flit interval (in clock cycles).
- Bursty traffic can also be realized. Bursty traffic is represented by alternating on and off periods. During the on period, packets are generated in fixed intervals. During the off period, no packets are generated. Traffic can be described in the following exponentially distributed variables: burst length (the number of packets in a burst on period) and off time (interval between two bursts). Developers need to specify the packet size (in bytes), load percentage (percentage of channel bandwidth to be used), destination (a fixed destination or randomly chosen destination), inter-flit interval (in clock cycles), average burst length (in number of packets) and average off time (in number of clock cycles).
- Trace based traffic. By defining a suitable format for packets flowing in the NoC, the traffic generated by other architectural simulators and stored in files, such as memory accesses, can be read into the simulator to evaluate the performance of network-on-chip if such a network is used to connect different parts of a system. This allows to compare network-on-chip performance for different sets of parameter choices for the same traffic.

3.6 Integrating Processor Cores with NoC Infrastructure

In this section, we construct a simulator, which models NoC-based manycore chips with a message-oriented distributed memory architecture that is commonly used in the embedded domain, to show the extensibility of our simulation framework. This simulator and related tool-chain provide tools for other research issues described later in this thesis.

The simulator is developed for NoC based homogeneous manycore chips supporting the distributed memory model which commonly exists in embedded systems. The manycore architecture has PowerPC 405 (PPC 405) cores embedded into tiles which are connected by a 2-D mesh. Inside each tile, a PPC405 core is connected to a network interface which contains several finite state machines to handle data transfers between cores and routers. Instructions are added to the PowerPC ISA to enable explicit communication between cores. Then several C APIs are provided based on which an MPI compatible library is implemented to enable high level programming. A cross-compiler tool chain is also created based on the GCC cross-tool for compilation of programs. With these supports, this simulator can run parallel applications written with a subset of MPI APIs. This simulator is used in our research work described in following chapters and therefore it is described in detail in this chapter. We also hope these details will be useful for others who would like to use our simulation framework.

Based on the literature review, an on-core module based interfacing strategy is adopted. Thereby, the architectural extension to PowerPC 405 core and added instructions are introduced first. Then the details of NI implementation are described. Following that, programming interfaces are designed to enable the simulator to run parallel applications. Finally, a simple MPI application is executed on this simulator and the various data captured for performance evaluation are elaborated. These highly accurate data manifest the capabilities of the simulators based on the simulation framework.

3.6.1 Microarchitecture

In this simulator, as shown in Figure 3.6 (a), multiple tiles are connected with a 2-D mesh. The size of mesh is configurable by the user. In this figure, the coordinates of the tiles at the left-top corner and the right-bottom corner are (0, 0) and (7, 7) respectively.

A PowerPC 405 core is incorporated into each tile and connected to the NoC through the NI, shown in Figure 3.6 (b). The PowerPC 405 is a single-issue, scalar core with a 5-stage pipeline. It has many features desired for future manycore chips [ABC⁺06]. More details related to the PowerPC 405 core are described in following subsections.

As this simulator has a distributed memory architecture, each tile can be regarded as an independent mini-computer. Therefore, each tile has a local private memory which is connected to a local bus. Other components inside a tile, such as a private cache, network interface,

Figure 3.6: An NoC-based homogeneous MPSoC with an 8×8 mesh

PowerPC 405 core, are also connected to the local bus. The network interface is the bridge of communication between tiles which will be discussed later. When data communications between the local tile and other tiles occur, the data can be transferred between network interface and memory (including cache) without involving the CPU.

Further, we have made some modifications to the cache module in order that 1) recently updated data in cache but not included in memory can be obtained by the network interface, 2) non-updated data in cache can be obtained from cache instead of memory to reduce the latency, and 3) data from network interface can also be updated into cache and memory to keep data coherence.

3.6.2 Interfacing The PowerPC 405 core with NoC

3.6.2.1 Architectural Extension to PowerPC 405 Core

The PowerPC 405 core is extended by interfacing the NI as an accelerator to it. In detail, a pair of input/output ports are added to PowerPC 405 core. The signals of these ports are of type *nireq*, a class wrapping the data passed via these ports. The core sends commands to and gets data from the connected NI through these ports.

A *nireq* signal contains the following: 1) a command passed to NI from PowerPC 405 core indicating to send/receive a message; 2) *id* of message specified by application; 3) tile id

of destination (source) for sending (receiving) the message; 4) the effective address of memory where the data of message to fetch (store) for sending (receiving); 5) length of message in bytes. Here, a simple message format is used. The first word (32 bits) of a message is the length of the message in bytes excluding the first word.

Figure 3.7: Format of added instructions

3.6.2.2 Extensions to instruction set of PowerPC 405

To expose network access operations to PowerPC 405 core, two instructions are added to PowerPC ISA. The mnemonics of them are “send” and “recv” respectively. They are designed in the same format with integer load/store instructions such as **lwz** and **stw**, shown in Figure 3.7. Opcode fields of “send” and “recv” are 57 and 58 respectively. Field *D* indicates a register and its content is denoted as (*D*). The higher 16 bits of the register content, i.e., (*D*), indicate *id* of the message to be sent or to be received. The lower 16 bits of (*D*) indicate *id* of destination tile for “send” and id of source tile for “recv”. Field *A* is a register and field *d* is an immediate. (*A*) + *d* indicates the message’s address in memory.

To support these instructions, it is necessary to modify the assembler to produce object codes. The back-end of the GNU assembler which is part of the binutils collection of tools is modified as follows. Two entries are added to the array of structure “powerpc_opcode”, i.e., powerpc_opcodes[], in file *opcodes/ppc-opc.c*. They are as follows:

```
{ “send”, OP(57), OP_MASK, PPC405, {RS, D, RA0} },
{ “recv”, OP(58), OP_MASK, PPC405, {RS, D, RA0} },
```

After the binutils tools are built, a new cross-compiler is generated using crosstool 0.43 [Keg09] for gcc 4.1.0 and glibc 2.3.6 which can generate object code for these instructions.

3.6.2.3 Functionalities of Added Instructions

During their execution, these two instructions “block” the pipeline of PowerPC 405 until they finish. When such an instruction finishes, the length of message sent or received is passed to PowerPC 405 core by the NI and this value is kept in the register D .

The “send” instruction is handled by the PowerPC 405 core like a store instruction. First, the core passes to the NI a *nireq* signal containing message id, destination id and the starting address of the message in memory. Then, the NI fetches the data of the message from the memory hierarchy and the NI sends them to NoC. After the whole message is sent, the NI returns the number of bytes sent to the core and the “send” instruction finishes.

The “recv” instruction is handled like a load instruction. Similarly, the core first passes to the NI a *nireq* signal containing message id, source tile id and the starting address to store the received message. Then, the NI checks for the arrival of expected message and moves its data to the specified memory address. After the whole message is sent to memory, the NI returns the number of bytes received to the core and the “recv” instruction finishes.

3.6.2.4 Finite State Machines in Network Interface

An NI contains two finite state machines (FSMs) for “send” and “recv” instructions respectively. When an NI starts to send a message to the NoC, the “send” FSM works as follows. 1) the NI first sends one memory request to the memory hierarchy to get the message’s head containing the length of the whole message. 2) After the length is obtained, successive memory requests are prepared and then sent to the memory hierarchy to get the whole message. 3) When replies to above memory requests come, the NI stores them in the buffer. 4) Each cycle, even before the whole message arrives, the NI tries to organize packets from its buffer and put them into an outgoing queue. The NI also tries to send to the NoC a flit of the packet at the head of the queue. 5) After the whole message is sent, NI returns the number of bytes sent to the core and the “send” FSM transfers to the idle state.

The “recv” FSM works in a similar manner. The difference lies in the reverse direction of data flow, i.e., from NoC to memory hierarchy. When packets come before the “recv” instruction which accepts them, they are kept in the buffer of NI if empty slots are available. When the buffer of NI is fully occupied, flits of packets are kept in buffers in routers and the flow control mechanism eventually takes in action.

In NoC designs, since that data that a packet accommodates may not be the same size as a memory request, the alignment of data is unavoidable. Additionally, in order that NI works properly as above described, some modifications have been made to the cache module and bus module provided by UNISIM for the purpose of cache coherence.

3.6.3 Programming interfaces

Several levels of programming interfaces have been designed to make programming easier. Interfaces at higher level are dependent on the ones at lower levels. They are described in detail as follows.

3.6.3.1 Low-level C Wrapper Functions

The “send” and “recv” instructions are wrapped in two C functions “_send” and “_recv” as below. Assembly intrinsics of “send” and “recv” are embedded in C codes and thereby only modifications to GNU assembler are needed.

```
int _send(int msg_tile_ids, char* message) { asm ( “send 3, 0(4)”); }  
int _receive(int msg_tile_ids, char* message) { asm ( “recv 3, 0(4)”); }
```

For these functions, it is necessary to understand procedure interfaces and register conventions for PowerPC processors. In PowerPC assembly, general purpose register GPR_n is represented by number n . According to [SHW96], GPR_1 is used as stack pointer register (SP). GPR_3 , GPR_4 , $\dots$, are used to store the first, second, $\dots$, parameter passed from caller of a function. Register GPR_3 is also used to store the first word of the return value if the function has return value.

In the “_send” function, msg_tile_ids contains a mixture of message id and destination tile id and $message$ is the address of the message to be sent. These parameters are passed to GPR_3 and GPR_4 respectively. The “send” instruction can get desired values. Similarly, in the “_receive” function, msg_tile_ids contains mixture of message id and source tile id. The $message$ parameter is the address to save the message.

Note that no *return* statement appears in above functions. In this way, a value set by instruction “send” or “recv”, i.e., length of message, is intact and returned to caller via register GPR_3 .

3.6.3.2 High-level C Wrapper Functions

Two high-level C wrapper functions, “Send” and “Receive”, are added to release the burdens of using low-level functions such as calculating *msg_tile_ids*, preparing messages in the correct format and allocating buffers, etc.

<pre> int Send(int msgId, int tileId, char* msg, int len) //msgId is a unique identifier of a message to be sent; //tileId is a unique identifier for the destination tile; //msg is the address where the message is stored; //len is length of the message in bytes; { char *temp, *t; int iLen = len; int msg_tile_ids = msgId << 16 + tileId; temp = (char*)malloc((iLen + 5)*sizeof(char)); if(temp == NULL) { return 0; } memset(temp, 0, iLen + 5); temp[0] = ((iLen & 0xff000000) >> 24); temp[1] = ((iLen & 0x00ff0000) >> 16); temp[2] = ((iLen & 0x0000ff00) >> 8); temp[3] = ((iLen & 0x000000ff)); t = &temp[4]; memcpy(t, msg, len); iLen = _send(msg_tile_ids, temp); free(temp); return iLen; } </pre>	<pre> int Receive(int msgId, int tileId, char* msg, int len) //msgId is a unique identifier of a message to be received; //tileId is a unique identifier for the source tile; //msg is the address where the message to be stored; //len is length of the message in bytes; { char *temp, *t; int iLen = len + 32; int msg_tile_ids = msgId << 16 + tileId; temp = (char*)malloc(iLen * sizeof(char)); if(temp == NULL){ return 0; } memset(temp, 0, iLen); iLen = _receive(msg_tile_ids, temp); memcpy(msg, temp, iLen); msg[iLen] = 0; free(temp); return iLen; } </pre>
--	--

Table 3.1: High-level wrapper functions

Table 3.1 shows these functions which include the calculation of *msg_tile_ids*. Buffers for temporary store are managed as well. In addition, in the “Send” function, the length of a message is calculated and placed at the beginning of the message. In “Receive”, an extra 32 bytes are allocated to save an extra memory request in NIs for non-aligned data.

3.6.3.3 Message-Passing Interface (MPI)

MPI is a standard interface for message-passing [MPI09] that supports point-to-point communications and collective operations. Programs using MPI typically employ single program, multiple data (SPMD) parallelism. MPI APIs are usually organized in layers and a layered implementation is adopted in this simulator. Point-to-point operations, *MPI_Send* and *MPI_Recv*,

are implemented based on above “Send” and “Receive”. Similar to [SC06], MPI collective operations such as *MPI_Bcast*, *MPI_Reduce*, *MPI_Barrier* and *MPI_Gather* are implemented on top of point-to-point operations using a linear algorithm. The details of *MPI_Send* and *MPI_Recv*, are as follows:

```
void MPI_Send (void *value, int len, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
{
    int size, destTileId, iUnitSize, iLen, iMsgId = 0, iReturn = 0;
    MPI_Comm_size(comm, &size); //to get the number of processes
    assert(dest < size); // make sure that dest is legal
    destTileId = grp_array[comm].member[dest]; //grp_array manages groups of processes.
    //iUnitSize is the size of data unit which is variable for different data types
    if ( type == MPI_INT ){
        iUnitSize = sizeof(int);
    }
    else if ( type == MPI_FLOAT ) {
        iUnitSize = sizeof(float);
    }
    else if ( type == MPI_DOUBLE ) {
        iUnitSize = sizeof(double);
    }
    else {
        printf(“MPI_Send: datatype is unknown”);
        return;
    }
    iLen = len * iUnitSize; //total length of data in bytes
    iMsgId = tag;
    iReturn = Send(iMsgId, destTileId, value, iLen); //send message after parameters are set
}
```

```
void MPI_Recv(void *val, int len, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Status
```

```
*status)
{
    int size, srcTileId, iMsgId = 0, iReturn = 0;
    MPI_Comm_size(comm, &size);
    assert(src < size);
    srcTileId = grp_array[comm].member[src];
    iMsgId = tag;
    iReturn = Receive(iMsgId, srcTileId, val, len); //receive data after parameters are ready
    //status is used to store the status information
    status->tag = tag;
    status->source = src;
    status->length = iReturn;
}
```

The MPI environment is setup in API *MPI_Init*. In it, each MPI process reads a configuration file whose name is passed from the command line. This file contains the information of one-to-one mapping between tile ids and ranks of processes related to a communicator. Currently, only communicator *MPI_COMM_WORLD* is used. APIs *MPI_Comm_rank* and *MPI_Comm_size* are based on above information as well.

3.6.4 Evaluation of an MPI based parallel program

A Linux OS plugin provided by UNISIM, *PowerPCLinux*, is attached to the PowerPC 405 core in each tile. These plugins can load ELF32 files compiled and statically linked to UNISIM libraries by the aforementioned cross-compiler. A configuration file is used to pass command line parameters to each core. Parameters for an application include local tile id, MPI configuration file and log file for debugging. Cores which run the same parallel workload are synchronized for termination of simulation. Upon termination of simulation, the detailed data of flits and messages are saved into files on the hard disc.

To elaborate the accuracy of the captured data, a parallel program using MPI to calculate the value of PI (π) [PI09] is evaluated on the simulator. In it, the root process (*rank* = 0) accepts the number of steps of calculation (*n*) from console and broadcasts it using *MPI_Bcast* to all non-root processes associated with the same communicator. Then, partial results from non-root

processes are collected using *MPI_Reduce* and added at root process to get the final result for above n . The program finishes when root process gets the zero value of n ($n = 0$) and non-root processes get this value from root process.

A simulator with 9 tiles arranged as 3×3 mesh is generated. Its parameters include:

- Sizes of level-1 instruction and data caches are 8K respectively.
- Bus buffer depth is 16.
- Routing algorithm is X-Y routing.
- NI buffer depth is 16.
- The number of virtual channels is 2.
- Virtual channel buffer depth is 8.
- Flit size is 64 bits.

In simulation, all 9 cores participate in the calculation and each core runs a process. The root process runs on a tile at the top-left corner, shown in Fig. 3.8. The mapping of processes to tiles is saved in a configuration file which is used by *MPI_Init*. The time in cycle when simulations start is set as 0. Two consecutive inputs to number of steps n are 50 and 0.

Figure 3.8: Root process runs on a tile at corner

The execution of the program takes 948921 cycles. The value of PI (π) is calculated as 3.1416258780877069 and the relative error compared to a predefined value is 0.0000332244979138. Part of the experimental data on flits and messages are presented in Table 3.2 and 3.3 respectively. Since each message is contained in a packet, information of packets is omitted.

3.6.4.1 Information on Flits

The simulator records the important events in the whole life of any flit. The first column of Table 3.2 contains partial identity information of flits. Each flit is labeled with its source tile id, destination tile id, tag of the message to which it belongs and the serial number of it in the message. “Generated Cycle” contains the time in cycle when the flit is created by its source NI and “Consumed Cycle” contains the time in cycle when the data of it is handled by destination NI. The combination of “Flit” and “Generated Cycle” can be used to uniquely identify a flit. The “NoC Entry Cycle” indicates the time in cycle when a flit leaves the source NI for the local router. The “NoC Exit Cycle” indicates the time in cycle when a flit leaves a router and enters the input channel of destination NI. Table 3.2 lists information of six flits. The first / last three flits are for the communications between root process and process 1 ($rank = 1$) / process 8 ($rank = 8$).

Table 3.2: Relevant details of some flits in the simulation

Flit (s,d,t,f)	Generated Cycle	NoC Entry Cycle	NoC Exit Cycle	Consumed Cycle
(0,1,0,0)	167744	167745	167761	167762
(1,0,0,0)	259214	259215	259231	480429
(0,1,0,0)	698098	698099	698115	698116
(0,8,0,0)	406135	406136	406177	406178
(8,0,0,0)	493480	493481	493521	611577
(0,8,0,0)	935986	935987	936027	936028

Note: s: source tile id; d: destination tile id; t: message tag; f: flit id.

3.6.4.2 Information on Messages

Messages in the simulation are displayed in Table 3.3. Meanings of columns of the table are as follows. The first column gives the partial identity information of a message which includes the message’s source tile id, destination tile id and tag. “Send Start” indicates the time (in cycles) when the PowerPC 405 core executes a “send” instruction and sends a command to source NI to initialize a sending session. “Send End” is the time in cycle when the NI finishes sending the message and returns value to the core. “Recv Start” and “Recv End” have the similar meanings but for a “recv” instruction. Length of a message is represented by bytes and flits, as shown in column “Length”.

During the simulation, there are 24 messages transferred in the NoC. Eight messages containing value of n (50, 4-byte integer) are sent from root process to non-root processes, listed in

Table 3.3: Relevant details of all messages in the simulation

Msg	Send (cycle)		Recv (cycle)		Length	
(src, dst, tag)	Start	End	Start	End	bytes	flits
(0,1,0)	167739	167746	141578	167764	4	1
(0,2,0)	201778	201785	141279	201812	4	1
(0,3,0)	235755	235762	141313	235780	4	1
(0,4,0)	269954	269961	141546	269988	4	1
(0,5,0)	303978	303985	141403	304020	4	1
(0,6,0)	337996	338003	141555	338030	4	1
(0,7,0)	372055	372062	141285	372096	4	1
(0,8,0)	406130	406137	141739	406180	4	1
(1,0,0)	259209	259216	480423	480431	8	1
(2,0,0)	293089	293096	499276	499284	8	1
(3,0,0)	327055	327062	518013	518021	8	1
(4,0,0)	361385	361392	536752	536760	8	1
(5,0,0)	391083	391090	555368	555376	8	1
(6,0,0)	425227	425234	574111	574119	8	1
(7,0,0)	459150	459157	592908	592916	8	1
(8,0,0)	493475	493482	611571	611579	8	1
(0,1,0)	698093	698100	292207	698118	4	1
(0,2,0)	732258	732265	326144	732291	4	1
(0,3,0)	766428	766435	360124	766453	4	1
(0,4,0)	800515	800522	394383	800548	4	1
(0,5,0)	834477	834484	424122	834518	4	1
(0,6,0)	868314	868321	458135	868347	4	1
(0,7,0)	902106	902113	492019	902148	4	1
(0,8,0)	935981	935988	526439	936030	4	1

Note: Msg: message; src: source tile id; dst: destination tile id; tag: message tag.

the first 8 rows of Table 3.3. Eight messages containing the partial results (8-byte double) are sent from non-root processes to root process, shown in the second 8 rows of Table 3.3. Eight messages from root process to non-root processes containing $n = 0$ are displayed in the last 8 rows.

3.6.4.3 Discussions

Based on the data shown in Table 3.3, delay and overheads of messages can be calculated (this is achieved by the simulator) and these results are shown in Table 3.4.

Delay of a message is the difference of the “NoC Exit Cycle” of its tail flit and the “NoC Entry Cycle” of its header flit, i.e., the period of time that a message stays in NoC. According to definition in Section 3.4.5, “total overhead” can be calculated as (“Recv End” - “Send Start” - “Delay”). “Total overhead” can be further divided into “Send Overhead” and “Recv Overhead”

Table 3.4: Delay and overheads of all messages in the simulation

Msg	Delay	Overhead (cycles)			SynCost
(src, dst, tag)	cycles	Total	Send	Recv	cycles
(0,1,0)	16	9	6	3	26183
(0,2,0)	25	9	6	3	60530
(0,3,0)	16	9	6	3	94464
(0,4,0)	25	9	6	3	128439
(0,5,0)	33	9	6	3	162614
(0,6,0)	25	9	6	3	196472
(0,7,0)	32	9	6	3	230808
(0,8,0)	41	9	6	3	264438
(1,0,0)	16	221206	6	221200	221192
(2,0,0)	24	206171	6	206165	206157
(3,0,0)	16	190950	6	190944	190936
(4,0,0)	24	175351	6	175345	175337
(5,0,0)	32	164261	6	164255	164247
(6,0,0)	24	148868	6	148862	148854
(7,0,0)	33	133733	6	133727	133719
(8,0,0)	40	118064	6	118058	118050
(0,1,0)	16	9	6	3	405908
(0,2,0)	24	9	6	3	406144
(0,3,0)	16	9	6	3	406326
(0,4,0)	24	9	6	3	406162
(0,5,0)	32	9	6	3	410393
(0,6,0)	24	9	6	3	410209
(0,7,0)	33	9	6	3	410126
(0,8,0)	40	9	6	3	409588

Note: Msg: message; src: source tile id; dst: destination tile id; tag: message tag; SynCost: synchronization cost.

where “Send overhead” is calculated as (“NoC Entry Cycle” of header flit - “Send Start”) and “Recv Overhead” is calculated as (“Recv End” - “NoC Exit Cycle” of tail flit).

Two important issues need to be addressed at this point. First, captured data are cycle accurate and accurate performance metrics can be calculated easily with these data. Metric Throughput (TP) can be obtained based on information on flits. Metrics delay and overhead can be calculated using information on messages.

Second, analysis on these data indicates a new metric is helpful to performance evaluation of coupled simulation. A study on “Overhead” in Table 3.4 reveals that the middle 8 rows have much higher values of “Total Overhead” than other 16 rows. These values come largely from “Recv Overhead” which is caused by the fact that “Recv” instruction to accept the message comes later than the “Send” instruction sending this message. After such a message is sent, it

waits for the corresponding “Recv” instruction in buffers of NI in the destination tile or buffers of some routers. By contrast, those small values of “Total Overhead” are resulted from the fact that the “Recv” instruction to accept a message comes earlier than the corresponding “Send” instruction. The cores executing “Recv” instruction wait for the desired messages to arrive.

Disadvantages exist for both above situations. For situation where messages wait for acceptance in NI or buffers in routers, these idle messages hold and waste the limited and precious resources for long periods. For situation where cores executing “Recv” instructions wait for desired messages, the compute cycles are wasted without doing any useful work. A new metric, namely *synchronization cost*, is proposed to quantify these disadvantages. The collected synchronization costs of messages are displayed in the last column, i.e., “SynCost”, in Table 3.4. When synchronization costs are reduced, performance can be improved, or resources can be better used, or both can be achieved. Therefore, synchronization cost can be used as an indicator for performance and resource-utilization efficiency.

3.7 Summary

We have developed a simulation framework for constructing simulators for NoC based multi-/many-core processors. Using it, we have also presented a simulator for embedded NoC-based homogeneous manycore systems containing PowerPC 405 cores. The corresponding tool-chain is generated such that real parallel applications can be compiled and executed on the simulator. Finally, an MPI based parallel program is evaluated on the simulator and the captured data show that performance evaluation of simulations can be of high accuracy.

The major features of our simulation framework are as follows:

- The framework has an on-chip network built upon pipelined routers which have a modular architecture and support wormhole switching and virtual-channel flow control. Various parameters of on-chip network, such as dimension, sizes of flit and packet, routing algorithms, etc., are configurable and performance metrics such as throughput, delay and overhead can be captured. By changing the connections between routers, the on-chip network can have different topologies.
- The framework is designed as a foundation for building simulators for manycore chips with different architectures. Using suitable core interfacing strategy, various cores can

be connected to NoC via network interfaces and network access operations are explicitly exposed to cores. Communicating with each other, different parts of parallel applications can be distributed and run on cores.

- The framework can be used to construct simulators which support typical parallel programming models such as message-passing and shared-memory.
- The framework is developed using the UNISIM environment. Many independent modules composing the simulator framework are connected through signals and these modules can be easily changed or replaced by others, enabling component re-use, fast customization and easy upgrading.
- The framework is developed as a cycle-level model of UNISIM which is characterized by high accuracy performance evaluation comparing to the real hardware and it helps the exploration of fine-grained parallelism.

3.7.1 Novelty of Our Research

Compared to simulators mentioned in Section 2.3.2, our work on the simulation platform for NoC based manycore systems has the below novelties: 1) different from simulators such as NNSE, Noxim, NIRGAM, Orion and LUNA, the simulators generated using our simulation framework support coupled simulation of both CPU cores and NoC; 2) unlike GEMS which supports shared-memory programming model and uses aggressive out-of-order CPU cores, the simulators generated using our simulation framework support message-passing programming model and utilize small and energy-efficient CPU cores which are desired for manycore systems.

Chapter 4

Accelerating Micro-architectural Simulations on Multicore Platforms

4.1 Introduction and Motivation

As shown in the literature review, “manycore era” is fast approaching. A manycore chip is a large-scale VLSI circuit that has many cores on a single die. Intel demonstrated its 80-tile manycore prototype connected via network-on-chip [VHR⁺07]. Tilera produces processors with dozens or hundreds of cores for embedded applications [Til09].

Simulators have been widely used for evaluating performance of processor designs without building costly physical hardware. But the complexity of manycore chips, multiple performance metrics and functionalities make simulator development for these chips a huge challenge. UNISIM [ACG⁺07], an environment built atop of the standard SystemC, has been proposed to facilitate simulator development by enabling reuses of components and control logic. Being successfully applied in building cycle-level simulators [Cel09] [UNI09], UNISIM has been shown to ease the development of simulators for multiprocessor systems.

However, despite the appealing benefits, UNISIM cycle-level simulations for multi-/many-core systems can be very slow because of the large number of modeled hardware modules and the high computing cost for simulating their activities. This slowness significantly restricts the design space that can be explored in affordable time. It could also limit the wider spread of UNISIM among researchers. Therefore, suitable techniques that can accelerate these simulations not only enable more design space exploration but also possibly raise more interest in UNISIM.

Nowadays, multicore computers are prevalent for speedup through on-chip, thread-level parallelism. But UNISIM cycle-level simulations are not scalable and cannot utilize multiple

cores for speedup. They are driven by a fast SystemC engine that is evolved from the OSCI SystemC engine [OSC09] by using acyclic scheduling [PMT04]. Like the OSCI engine, the UNISIM engine is single-threaded. Hence, we are motivated to accelerate these simulations on multicore platforms by exploiting parallelism. However, it is quite challenging to parallelize a cycle-level simulation because such a simulation is essentially sequential and can only be advanced cycle by cycle. Methods used to parallelize other single-threaded code by exploiting coarse-grained parallelism cannot be applied to such simulations.

To explore parallelism, we study a typical cycle-level simulation of a microarchitectural simulator for manycore systems based on UNISIM (described in Section 4.8). Read Time-Stamp Counter (RDTSC) instructions are used to accurately measure the time for simulating each cycle on a simulation host without measurable performance impact. We observe two facts: 1) the total computation of executing SystemC processes in some cycles is enough to run multiple threads efficiently. However, though processes of different modules can be run concurrently, the UNISIM engine handles them sequentially. 2) Computation of individual processes and the total computation vary in simulated cycles due to the dynamic behavior of the simulated system. These facts present opportunities for achieving speedups by realizing parallel simulations.

In this chapter, we present a systematic framework that aims at accelerating single-threaded UNISIM cycle-level simulations on multicore platforms by exploiting parallelism. First, we propose to transform the single-threaded UNISIM engine into a multithreaded engine so as to facilitate exploitation of fine-grained parallelism within the simulated cycles using threads. Moreover, each cycle is decomposed into several sequential and parallel sub-steps. In parallel sub-steps, threads run concurrently. SystemC objects of a simulated system are distributed into disjoint partitions and simulation of each partition is run by a dedicated thread. Thus, original single-threaded simulations are carried out by multiple threads concurrently.

Second, we analyze runtime behaviors of threads in multithreaded simulations by looking at sub-steps of simulated cycles. We observe that better balanced workloads among threads can lead to higher performance and computation variations have a negative influence on load balancing of threads. We further obtain the conditions for achieving acceleration when a single-threaded simulation is replaced by an equivalent multithreaded simulation where the activities of the simulated system remain the same but are carried out by multiple threads, rather than by a single thread.

Finally, to overcome the challenge of computation variations, we accomplish adaptive simulations via the proposed multithreaded engine. The engine checks overall computation of a multithreaded simulation periodically. After detecting large variations of computation, the engine adjusts the number of threads employed in the simulation to satisfy conditions for acceleration to achieve speedup. It further balances workloads of threads by distributing SystemC objects among partitions handled by threads. The partition graph of a simulated system is introduced and manipulated by the engine to automate the distributions of SystemC objects.

4.1.1 Contributions and Chapter Organization

Contributions of this chapter are as follows:

- a novel method is proposed to exploit fine-grained parallelism within the simulated cycles to accelerate SystemC cycle-level simulations;
- a realization of a multithreaded engine is presented to facilitate the above mentioned exploitation of parallelism;
- techniques are developed to achieve high performance by adapting multithreaded simulations to computation variations;
- most importantly, the systematic framework for acceleration proposed in this chapter can be generalized to any discrete event simulation engine with delta-delay semantics.

The rest of this chapter is organized as follows. Section 4.2 presents the motivation for multithreaded cycle-level simulations. Section 4.3 describes the main ideas and the overview of our proposed framework. The following sections 4.4, 4.5, 4.6 and 4.7 discuss techniques within the framework in depth. Section 4.8 gives details of experiments and results. Section 4.9 concludes this chapter.

4.2 Motivation for Multithreaded UNISIM Cycle-level Simulation

In this section, we provide some background about UNISIM cycle-level simulation and explain the motivation for realizing multithreaded simulations via an example.

4.2.1 Overview of Single-threaded UNISIM Cycle-level Simulation

The UNISIM engine, similar to the OSCI SystemC engine, manipulates SystemC objects, such as *modules*, *ports*, *signals* and *processes*, in the executable of a simulator. A simulator consists of *modules* for hardware components which have variables to store their individual states and methods for implementing their functionalities. Usually, each module has *ports* which are used to accept inputs or transfer outputs to other modules. One output port of a module is connected to a *signal* which itself is connected again to an input port of another module. For system behavior, a simulator typically includes *sequential processes* which are sensitive to a clock edge and *combinational processes* which are sensitive to their input ports. SystemC processes and ports are always contained within certain modules.

A simulated cycle consists of two clock phases that start with rising and falling edges respectively. At each clock phase, the UNISIM SystemC engine follows simulation semantics, i.e., the following steps:

- (i) Processes sensitive to the clock edge are first woken up. During their execution, they modify outgoing signals.
- (ii) The signals that have been changed are updated.
- (iii) The processes with sensitive input signals that have changed are executed.
- (iv) Step 2 and 3 are repeated until no more process needs to be waken up. One iteration including Step 2 and 3 is known as a delta cycle.
- (v) Simulation is advanced to next clock edge.

The UNISIM engine maintains pointers to SystemC objects in several list-like tables, namely T_{seq} , T_{sig} and T_{com} , which are global variables. Table T_{seq} is used for storing sequential processes. It is filled when a simulator is initialized and keeps intact during simulation. Table T_{sig} is for signals that are modified during execution of both sequential and combinational processes. Table T_{com} is for the combinational processes which are sensitive to the modified signals in T_{sig} and will be executed in the current delta cycle. T_{sig} and T_{com} are filled dynamically. When a signal s is modified at Step 1 or 3 of above semantics, s is added into T_{sig} . At Step 2, processes sensitive to the signals in T_{sig} are added into T_{com} . At Step 3, updated values of signals are read by processes.

4.2.2 Exploring Fine-grained Parallelism

Figure 4.1: Time for simulating each cycle of a simulation on host computer

We study a cycle-level simulation run on a UNISIM based microarchitectural simulator for a 64-tile Network-on-Chip (NoC) based MPSoC whose tiles are connected as an 8×8 mesh. It has a large number of SystemC objects: 2,592 modules, 13,408 processes and 21,433 signals. Its details are described in Section 4.8.

We use Read Time-Stamp Counter (RDTSC) instructions to accurately measure the time for simulating cycles on the host with ignorable costs. The *Time Stamp Counter* (TSC) is a 64-bit register present on all X86 processors since the Pentium. This counter has, until recently, been an excellent high-resolution, low-overhead way of getting CPU timing information and counts the number of ticks since reset. An RDTSC instruction returns the value of TSC in registers *EDX* and *EAX*, in the form of *EDX* : *EAX*. To evaluate the time of executing a SystemC process, two RDTSC instructions are placed at the beginning and the end of the source code of the SystemC process to get two time-stamp readings, which are used to calculate the time taken to execute the process. This measurement has a very low cost as it involves only two RDTSC instructions and one subtraction between two 64-bit integers. As the simulation time is mainly spent on executing invoked SystemC processes, the time for simulating a cycle can be approximated by the sum of the time used in executing these processes. The time for simulating each cycle is recorded and plotted in Figure 4.1, where the x and y coordinates of a point indicate a simulated cycle and the time for simulating the cycle respectively.

We obtain two interesting observations from this experiment. First, at each simulated cycle, the amount of computation of an individual SystemC process is commonly too small to sustain

the efficient running of a thread. However, for a simulator with a large number of SystemC processes, the aggregate computation of processes in some simulated cycles is possibly large enough to run several threads efficiently.

As the UNISIM engine accesses the pointers of processes stored in tables T_{seq} and T_{com} , execution of sequential processes in Step 1 and combinational processes in Step 3 is realized with loops where processes are accessed and run sequentially, i.e., one by one. However, due to inherent concurrency of modeled hardware, processes in different SystemC modules are independent from each other and can be executed in parallel when invoked at the same time. We can exploit this fine-grained parallelism within simulated cycles to realize parallel simulations using threads.

The second observation is that simulation computation of individual modules in simulated cycles fluctuates during simulation. So does the overall computation of all modules, shown in Figure 4.1. Processes have various amounts of computation due to their different functionalities. Moreover, the number of invoked processes and their execution paths could be different due to the dynamic behaviors of the simulated system, which result from interactions between its micro-architecture and software applications run on the micro-architecture.

If multiple threads are used to explore parallelism, a question is how much computation should be allocated to each thread. Intuitively, we could distribute computation evenly among threads. As there are varying amounts of computation in different simulated cycles, another question is how many threads should be used for a particular cycle. If computation in a cycle is limited and too many threads are used, the computation for a thread may be too small to support its efficient running, as there are always costs to start a thread. Otherwise, if the computation is large enough but too few threads are used, the chance for higher performance is missed. The ideal situation is that number of threads varies to match the changing computation. This is difficult since the exact amount of computation is unknown until simulation of the cycle is actually finished.

To sum up, it is possible to exploit fine-grained parallelism within original single-threaded cycle-level simulations to realize parallel simulations using threads for acceleration. But we have to overcome the challenge of varying computation.

4.3 The Proposed Systematic Framework

4.3.1 Ideas for Multithreaded UNISIM Cycle-Level Simulations

As discussed above, we can realize parallel simulations by utilizing fine-grained parallelism within simulated cycles, i.e., SystemC processes in different modules can be executed concurrently. Hence, we can divide all SystemC modules of a simulated system into disjoint partitions and a partition can have multiple modules. Accordingly, processes are put into partitions because processes are contained in modules. Then, simulation of modules within a partition, which includes executing processes and modifying signals, can be considered an independent computation strand to be run by a thread dedicated to this partition. In this way, multiple threads can be used to run simulations of partitions in parallel.

The amount of computation for a thread can be managed by adding (removing) modules into (from) its partition. Manipulating modules saves effort in keeping data integrity and thread safety when modules are distributed among partitions. A SystemC module is a basic unit of modeling and processes inside a module share data. Modules are connected by signals at ports. When modules are partitioned, signals can be easily categorized as “intra-partition” or “inter-partition” based on the information of SystemC objects available in a simulator.

When multiple threads drive a cycle-level simulation, one known challenge is to choose suitable numbers of threads for cycles in simulation. Another challenge of load balance is brought by the unavoidable synchronization of threads. An obvious example is that threads have to be synchronized to update the shared variable for the global clock cycle. Experience from parallel computing indicates that the efficient use of the multicore platform is achieved when cores are kept busy doing useful work and inter-core communication is kept at a minimum. When the workloads of threads are better balanced, the simulation can be completed in shorter time because less time is spent by some threads waiting for others before all can move on to next cycle. But the varying overall computation and thereby the varying computation in partitions in cycles makes it very difficult, if not impossible, to keep workloads of threads well balanced in such a parallel simulation.

Our ideas for overcoming the above challenges are to make a multithreaded simulation that adapts to the varying overall computation. Both overall computation of the simulation and the workloads of threads are checked periodically. Here, a *period* is defined in terms of simulated cycle and can be one or multiple cycles. If there are large changes of overall computation or

serious imbalances among workloads of threads between periods, the number of threads and their workloads are adjusted for better performance. This adaptive simulation is automated by a multithreaded engine which can be transformed from the original single-threaded UNISIM engine using POSIX threads.

The rationale for period based checking and adjustment is as follows. Given that length of periods is small enough, the amount of computation for simulating individual cycles within the same period should be close in many cases. Further, if the amount of computation in individual cycles is distributed nearly equally among threads, the workloads of threads should be similar too. Moreover, when the length of periods is small enough, most large variations of computation can be captured and the simulation can be adjusted promptly.

4.3.2 The Proposed Framework

In this section, we present the high level overview of our proposed framework with the help of the flowchart shown in Figure 4.2. We describe how the key techniques of the framework cooperate to make multithreaded simulation adaptive to the variations of computation during simulation and achieve high performance.

The first technique is proposed in Section 4.4 for transforming the single-threaded UNISIM SystemC engine using POSIX threads into a multithreaded engine that facilitates exploitation of fine-grained parallelism within simulated cycles.

When a simulation starts on a host computer with N physical cores, the proposed multithreaded engine creates N threads for realizing a parallel simulation. The engine divides SystemC objects into N disjoint partitions and each created thread is dedicated to run the simulation of a partition. When a simulation just starts and information of overall workloads is not available, the engine tries to utilize all cores for highest performance. This is done in the processes (1) and (2) of the flowchart.

Thus, a single-threaded simulation is replaced by its equivalent parallel simulation where activities of the simulated system remain the same but are carried out by multiple threads. The multithreaded simulation is advanced cycle by cycle, shown at process (3) in the flowchart, until the end of a period is met. A period consists of multiple simulated cycles and its length is taken by the engine as a parameter.

However, a single-threaded simulation is not guaranteed to be accelerated simply by using multiple threads. Therefore, at the end of a period, indicated by decision (4) in the flowchart, the

Figure 4.2: Overview of the proposed technique

engine checks if the simulation in the current period has been accelerated based on a technique that are discussed in Section 4.5.

Then, the engine decides, or more precisely predicts, the number of threads to be used in the next period. Here, the engine predicts activities of simulation in the near future, i.e., the next period, based on the recent past, i.e., the just finished period. This is rational based on the *Principle of Locality* as a simulator has temporal and spatial localities similar to those of the simulated system the simulator mimics.

As shown in process (5) of the flowchart, one possibility is that the number of threads for the next period is adjusted. The technique for adjustment is discussed in Section 4.7. After the number of threads is changed, so is the number of partitions. The engine has to redistribute SystemC objects to form new partitions.

The other possibility is that number of threads used in simulation remains same. As shown in decision (6) of the flowchart, the engine further checks if the workloads of threads are well balanced. If not, the engine redistributes the SystemC objects among partitions so as to keep the workloads of threads better balanced, shown in process (7) of the flowchart. These steps are discussed in Section 4.7 in detail.

In processes (5) and (7) of the flowchart, the engine redistributes the SystemC objects among partitions for better load balancing among threads. To automate the redistribution, we introduce a graph partitioning based technique in Section 4.6.

After the necessary processing is finished at the end of a period, the simulation proceeds to start a new period if the simulation has not yet completed.

We discuss our techniques in depth in the following sections. In Section 4.4, we describe how to obtain the proposed multithreaded engine. We discuss issues including subsections of a clock phase, categories of threads and how to keep to SystemC simulation semantics and maintain thread safety. In Section 4.5, we discuss behaviors of threads within subsections of simulated cycles in simulation and conditions for achieving acceleration. In Section 4.6, we define the partition graph of a simulated system and describe how the engine manipulates it at runtime for load balancing among threads. In Section 4.7, we present how the multithreaded engine monitors variations of overall computation and detects serious workload imbalances among threads. We further propose two strategies for adaptive multithreaded simulations.

Our framework is applied at the SystemC engine level. So, there is no need to modify the source code of simulators to utilize it. It also releases developers from the burden of programming threads. The above features make it different from existing work for accelerating SystemC simulations described in the literature review. Some existing techniques don't leverage parallel techniques. Other existing techniques transform the source code of the simulators and even modify elements of the SystemC language. Thus, they are not suitable for micro-architectural simulators that capture detailed activities of modeled hardware and whose source code cannot be modified.

Though currently our framework is based on the UNISIM engine, the scope of its application is potentially broader. First, it is applied at the level of the simulation engine and suitable for any simulator based on UNISIM. Second, as the UNISIM engine is evolved from the OSCI SystemC engine [PMT04] and our framework does not depend on any UNISIM specific features, it can be extended for application to OSCI engine. Finally, the framework enables scalability due to it being capable of generating and using the desired number of threads based on available resources and simulation computation. It is portable as it relies on general tools and platforms.

4.4 Parallelizing The Single-threaded UNISIM SystemC Engine

In this section, we present how to transform the single-threaded UNISIM engine into a multi-threaded engine by modifying its source code and using POSIX threads.

Figure 4.3: Sequential and parallel subsections in a clock phase

4.4.1 Sequential and Parallel Sections within A Clock Phase

To exploit fine-grained parallelism, each clock phase is divided into several sequential and parallel sections, shown in Figure 4.3. In realizing this, we modify the source code for a clock phase of the original UNISIM SystemC engine using POSIX threads. In parallel sections, such as PS_1 and PS_2 , multiple threads run simulations concurrently. In sequential sections, such as SS_0 , SS_1 and SS_2 , they are synchronized and only one of these threads runs to update global variables or exchange data. Sequential sections are protected by mutexes. Specifically, section SS_0 needs to complete work in the main loop of simulator. Depending on how the main loop is implemented, SS_0 can appear once or twice in a simulated clock cycle. We will discuss more about SS_0 later.

In addition, according to the Amdahl's Law, the sequential parts which cannot be accelerated by using parallelism should be kept minimal to achieve high performance. In realizing the multithreaded engine, we use efficient code in sequential sections, such as SS_1 and SS_2 . We also advise developers who use our proposed framework to keep overheads of code at the main loop, i.e., SS_0 , at a minimum. One example is to turn off debugging features, such as port connection checking, after simulators are ready to run in release mode.

4.4.2 Master Thread and Worker Threads in Parallel Simulation

Threads involved in parallel simulation consist of a master and multiple worker threads. The master thread is the one that OS uses to start a simulator. Worker threads are created by the engine (executed by master thread) when the simulator initializes. A condition variable for thread synchronization is created for each worker thread. This condition variable is used to suspend and wake up the thread that is associated with it. Worker threads and the master thread are kept alive until the simulation ends.

In order that our framework does not coerce any change on the single-threaded simulator development that is familiar to developers, only the master thread executes the code for the main loop at SS_0 . The master thread also runs through all other sequential and parallel sections. But it does not participate in simulation computation, i.e., executing SystemC processes, in PS_1 and PS_2 .

All worker threads are synchronized and blocked after each clock phase finishes. A thread sleeps until its condition variable is set by the master thread when the master thread runs at

SS_0 . Worker threads that are woken up continue to run in the next clock phase. In this way, the number of worker threads which are employed in the simulation in different clock phases can be controlled.

Assume that the SystemC objects of a simulator are divided into partitions, simulation of each partition is carried out by a worker thread. Thus, in PS_1 and PS_2 , several worker threads run concurrently aiming to improve simulation performance. Though the maximum number of worker threads can be arbitrary, we keep the maximum number at the number of physical compute cores. Thus, worker threads can be assigned to be executed on specific physical CPU compute cores, one thread per core, to save the costs of switching their thread contexts.

4.4.3 UNISIM SystemC Simulation Semantics

The SystemC simulation semantics mentioned in Section 4.2.1 are kept.

- (i) Sequential section SS_1 indicates the start of a new clock phase, i.e., the work of Step 5 of SystemC semantics.
- (ii) Sequential section SS_2 is where worker threads are synchronized to update modified signals. The work of Step 2 is done here.
- (iii) Calls to sequential processes sensitive to the current clock edge, i.e., work of Step 1, are run in parallel section PS_1 .
- (iv) Calls to the woken-up combinational processes, i.e., work of Step 3, are run in parallel section PS_2 .
- (v) Several iterations of SS_2 and PS_2 can happen depending on the modified signals. An iteration of SS_2 and PS_2 forms a delta cycle (Step 4).

4.4.4 Performance Optimization and Thread Safety in Multithreaded Simulations

We also make modifications to data structures used by the single-threaded UNISIM SystemC engine to achieve optimized performance using threads. We also maintain thread safety when these data structures are accessed by threads.

If global tables used by the single-threaded UNISIM engine, such as T_{seq} , T_{sig} and T_{com} mentioned in Section 4.2.1, are kept, they must be prevented from concurrent accesses by multiple threads. This can lead to large synchronization overheads.

Hence, assuming that the SystemC objects are already partitioned, we split these tables into smaller “partition tables” that are distributed into partitions. Specifically, assume that there are n partitions p^i where $1 \leq i \leq n$. A partition p^k has three *partition tables* t_{seq}^k , t_{sig}^k and t_{com}^k that contain the partial data, which are from T_{seq} , T_{sig} and T_{com} respectively, for partition p^k .

Ideally, in the parallel sections of a clock phase, if partition tables of a partition are accessed only by the worker thread dedicated to the partition, multiple threads can execute in parallel independently which greatly boosts performance.

However, some signals straddle between partitions and are thereby called *inter-partition signals*. They can be accessed by threads dedicated to different partitions. Such a signal is annotated as $s_{i \rightarrow j}$ if it is written by a SystemC process in partition p^i and read by a process in partition p^j where $i \neq j$. After $s_{i \rightarrow j}$ is written by the thread of p^i , it should be added into the partition table t_{sig}^j of p^j . Without proper handling, runtime thread safety becomes a concern and the simulation can crash.

We do analysis of thread safety as follows. In sequential sections of clock phases, there is no thread safety concern since only one thread executes. In parallel sections, threads safely access their partition tables t_{seq} and t_{com} concurrently. For any SystemC process, it is always executed by the thread of its partition. The process safely accesses data of its containing module, reads incoming signals and modifies outgoing signals which are connected to modules in the same partition. The thread safety violation can only occur when an inter-partition signal is modified. Therefore, inter-partition signals have to be handled specially to maintain thread safety.

Since there is no thread safety concern in sequential sections, we separate writing and submission of an inter-partition signal $s_{i \rightarrow j}$ in parallel and sequential sections respectively. $s_{i \rightarrow j}$ can be modified in a parallel section but has to be updated in the succeeding sequential section. For thread-safe storage, a new partition table is added to each partition to save the pointers of outgoing inter-partition signals modified by the partition’s thread in parallel sections. Such a table for p^i is annotated as t_{ip-sig}^i and “ip” in subscript stands for “inter-partition”. When $s_{i \rightarrow j}$ is modified in a parallel section, it is added into t_{ip-sig}^i of partition p^i . After threads are synchronized in the following sequential section, $s_{i \rightarrow j}$ is copied from t_{ip-sig}^i to t_{sig}^j of p^j . t_{ip-sig} tables

of all partitions are iterated to handle all of the modified signals. After inter-partition signals are added into their target t_{sig} tables, reading of their values in the following parallel section is thread-safe.

4.5 Accelerating Multithreaded Simulations

The multithreaded engine proposed in Section 4.4 paves the way for exploiting fine-grained parallelism. In this section, we study the activity of worker threads in subsections within clock phases. We also discuss conditions for achieving acceleration for periods of different lengths. We first define a *microcycle* which can be regarded as a fundamental representative element of cycle-level simulation.

Figure 4.4: Microcycles in a simulated cycle

4.5.1 Microcycle

Figure 4.4 shows two clock phases of a simulated cycle i . A *microcycle* is defined as a sequential section followed by a parallel section. A microcycle can consist of SS_1 and PS_1 . It can also be composed of SS_2 and PS_2 . A microcycle composed of SS_2 and PS_2 is actually a delta-cycle. A simulated cycle has multiple microcycles. As cycle-level simulation is composed of numerous simulated cycles, a microcycle can be regarded as a fundamental element of it. Conclusions from studying the behaviors of threads within a microcycle can be generalized to simulated cycles and even the whole simulation. Additionally, as SS_0 is executed by the master thread and it cannot be accelerated, the performance of simulation is mainly determined by activities of worker threads in microcycles.

4.5.2 Multithreaded Simulation within A Microcycle

To analyze the runtime behavior of threads, a single-threaded simulation in a microcycle and two of its equivalent multithreaded simulations are shown in Figure 4.5.

The single-threaded simulation is shown in Figure 4.5 (c). Two of its equivalent parallel simulations are run by 4 worker threads, shown in Figure 4.5 (a) and (b). In Figure 4.5 (a), thread t_2 is the last thread arriving at synchronization (A) and it carries out the computation in the sequential section (SC'). Then t_2 notifies OS to wake up other threads and continues the simulation of its partition (PC_2). After a while (LS), other threads t_1 , t_3 and t_4 are scheduled by OS to run simulations of their partitions respectively. Threads that finish their simulations earlier are suspended by OS (ES) to wait for other threads (W). In Figure 4.5 (a) and (b), threads t_4 and t_3 are the last ones arriving at synchronization (B) respectively.

Figure 4.5: Parallel simulations in a microcycle

By comparing (a) and (b) of Figure 4.5, it is obvious that better balanced workloads among threads lead to shorter simulation time, i.e., higher performance. Moreover, in an ideal situation where workloads of threads are distributed in a way that thread t_1 , t_2 and t_3 finish at the same time as t_4 and no thread waits, the time between B and A (denoted as $T(B) - T(A)$), i.e., the time for simulating a microcycle, is shortest.

4.5.3 Factors Affecting Acceleration

When multiple threads are used, overheads in parallel simulation mainly come from synchronization among threads. When the computation is small and distributed to an inappropriately

large number of threads, synchronization costs can counteract the benefits from parallel execution, even slowing down the simulation. Hence, it is important to choose a suitable number of threads to accelerate simulations.

To overcome varying computation in different periods during a simulation, the ideal situation is that number of threads can be automatically adjusted by the multithreaded engine to accommodate the varying computation to keep obtaining acceleration in periods. Thus, the engine should be able to evaluate conditions for achieving acceleration at runtime with low costs.

We first consider the condition for achieving acceleration in simulating a microcycle. Then we generalize it to periods whose lengths are larger than a microcycle. The informal expression of the condition targeting a microcycle is stated: *a single-threaded simulation in a microcycle is accelerated only if savings from parallel execution aggregately overcome overheads in its equivalent parallel simulation.*

However, we need to express the above condition in a form that can be easily evaluated by the multithreaded engine. For the microcycle in Figure 4.5, the time for the original single-threaded simulation (c) is longer than two equivalent multithreaded simulations (a) and (b). We study the condition for acceleration via this example.

Several relevant times are defined as below. 1) time of completing a microcycle in parallel simulation, i.e., $T(B) - T(A)$. 2) the time for running simulation PC_i , i.e., $T(PC_i)$, where $1 \leq i \leq 4$ for partitions. PC_i consists of executing SystemC processes of the modules which belong to the partition which thread t_i handles. Assuming that each core of a multicore host has same computing capability, this time remains the same in both a single-threaded simulation and its equivalent parallel simulations. 3) the time for executing simulation SC in single-threaded simulation, i.e., $T(SC)$. SC includes updating global variables, accessing tables to update signals and so on. So, the condition for acceleration in the microcycle in Figure 4.5 using 4 threads is:

$$\sum_{i=1}^4 T(PC_i) + T(SC) > T(B) - T(A) \quad (Ineq. 1)$$

We can easily obtain $T(B) - T(A)$ with low costs by placing two RDTSC instructions at proper places in source code. We could also obtain $\sum_{i=1}^4 T(PC_i)$ by adding up the time of executing SystemC processes (discussed in Section 4.2.2) which have been invoked in this microcycle. We don't need to obtain the times such as $T(SC')$ and $T(LS)$ since they are contained in

$T(B) - T(A)$. There is no way to obtain $T(SC)$ in parallel simulations and we have to leave it out. After omitting $T(SC)$ in *Ineq. 1*, we obtain a more demanding condition for achieving acceleration as follows:

$$\sum_{i=1}^4 T(PC_i) > T(B) - T(A) \quad (\text{Ineq. 2})$$

This condition can be generalized to periods of different lengths: multiple microcycles, a simulated cycle, multiple simulated cycles and even the whole simulation. We don't give out formal description for all periods. As the condition generalized to a simulated cycle will be used in the following sections, we discuss it as below.

At the source code for the beginning of a rising clock phase, we use RDTSC instructions to get the time for simulating a cycle using n threads, i.e., item $T(B) - T(A)$ in *Ineq. 2*. Item $\sum_{i=1}^n T(PC_i)$ can also be obtained using RDTSC instructions. The informal expression of the condition of achieving acceleration for a simulated cycle is: *In a simulation carried out by multiple threads, if the time for simulating a simulated cycle is less than the aggregate time used by these threads for executing all the SystemC processes invoked within this simulated cycle, the original single-threaded simulation in this simulated cycle is considered as accelerated.*

4.6 Partitioning for Load Balancing

As shown in Section 4.3, the multithreaded engine is proposed to achieve load balancing by distributing SystemC objects among threads based on the runtime workloads. Thus, we introduce a *partition graph* for the engine to automate the distribution of SystemC objects by applying a graph partitioning based technique.

4.6.1 Partition Graph of A Simulated System

A simulated system is represented as a weighted and undirected graph G , namely *partition graph*, where a SystemC module is represented by a vertex and the amount of computation of this module is taken as the weight of this vertex. A signal connecting two modules is represented by an undirected edge that connects the vertices of these modules. As any signal connecting different modules can be an inter-partition signal and all inter-partition signals are handled in the same way, each signal can be treated equally and thereby all edges of G are unweighted.

Figure 4.6: An example of partition graph, G

An example of partition graph is shown in Figure 4.6. The simulated system in this simple example has 16 vertices (modules), represented by ovals, which are connected by a mesh. The figures in the square brackets are the weights of vertices (computation of modules) respectively.

The multithreaded engine creates G using the information of SystemC objects in the simulator when a simulation starts. At that time, since information of runtime computation of modules is not available, weights of vertices are initialized to a default value. Then, the weights of vertices are updated by the engine at the ends of periods during simulation.

The aggregate time for executing processes of a module on a host computer during a period, which can be obtained by using RDTSC instructions, is taken as the weight of the module's vertex. With G , the *workload of a thread* is the sum of weights of vertices in the thread's partition and communications between two threads are in the proportion to the number of edges that straddle between these partitions.

4.6.2 Graph Partitioning Based Technique for Load Balancing

For high performance, we aim to keep workloads of threads balanced and the number of inter-partition signals minimum. We transform this problem into an equivalent graph partitioning problem P as below: dividing G of a simulated system into k disjoint partitions such that partitions have approximately equal weights and the number of edges that straddle partitions is

4.7.a: 3 partitions

4.7.b: 4 partitions

 Figure 4.7: Partition graph G with partitions

minimized. The *weight of a partition* is the sum of the weights of the vertices in the partition. Here, k is also the number of worker threads. Then, we solve P using a well known multilevel graph partitioning algorithm [KK98].

The results of dividing the partition graph G in Figure 4.6 into 3 and 4 partitions are shown in Figure 4.7 (a) and (b) respectively. The modules contained within a dotted closed line belong to a same partition. In Figure 4.7 (a), partition 1 has 5 modules: 1, 2, 5, 9 and 13; the weight of partition 1 is 318. Partition 2 also has 5 modules: 3, 4, 6, 7 and 8; the weight of partition 2 is 315. Partition 3 has the remaining modules and its weight of partition is 312. There are 8 inter-partition signals.

4.7 Deploying The Multithreaded Engine for Adaptive Simulations

In this section, we discuss how adaptive simulations are realized through the multithreaded engine. We first explain how the engine monitors the varying computation, checks whether simulation is accelerated by applying the conditions for acceleration and adjusts the number of worker threads when needed. Second, we describe how the engine detects and corrects serious imbalances among workloads of threads. Finally, we present two strategies for adaptive multithreaded simulations.

The code for realizing adaptive simulations is added at the beginning of the sequential section SS_0 (Figure 4.3) of the rising clock phase. The ending of a previous falling clock phase is logically equal to this place when there is no code between them. We use code of high efficiency to reduce overheads.

As defined before, a *period* consists of one or multiple simulated cycles. *Length of period (LOP)* is the number of simulated cycles within a period. In the discussions within subsections 4.7.1 and 4.7.2, i.e., before the strategies for adaptive simulations are introduced, it is assumed that a suitable value has been chosen for length of period in a multithreaded simulation.

4.7.1 Strategy for Accommodating Computation Variations at Runtime

Due to the dynamic behaviors of the simulated system, the overall computation of a simulation varies during simulation execution. If the number of worker threads employed in simulation is fixed, say m , this number could be inappropriate in two scenarios. One scenario is that the

overall simulation computation is too small to support running m worker threads efficiently. So, the simulation is decelerated as overheads counteract the benefits. The other scenario is that the computation is large enough to run more than m threads efficiently. With more threads, the simulation can further be accelerated. With the number of threads fixed, such chances can be missed. Thus, the number of worker threads in simulation should be adjusted to follow the varying overall simulation computation for better performance.

As discussed in Section 4.3, at the end of periods, the multithreaded engine predicts the number of worker threads for the next period, which is denoted as NWT_{next} . The number of worker threads in the current period is denoted as NWT_{curr} . When a simulation starts, NWT_{curr} is set to the maximum value, i.e., the number of physical cores on the multicore host, aiming to achieve high performance by using all cores.

We propose the following method to adjust the number of worker threads at runtime. The engine has a counter to record the number of cycles in a period that are judged as accelerated based on the condition for acceleration for a simulated cycle, which is similar to *Ineq.2* described in Section 4.5.3. The counter is used to help predict NWT_{next} as follows. 1) When the counter indicates that 100% cycles in a period are accelerated, we say this period is *strongly accelerated*. If two consecutive periods are strongly accelerated, $NWT_{next} = NWT_{curr} + 1$. If NWT_{curr} is the maximum value, $NWT_{next} = NWT_{curr}$. 2) When the counter indicates that less than 80% cycles in a period are accelerated, we say this period is *strongly decelerated*. If two consecutive periods are strongly decelerated, $NWT_{next} = NWT_{curr} - 1$. If NWT_{curr} is 1, $NWT_{next} = 1$. 3) Otherwise, $NWT_{next} = NWT_{curr}$. This method for predicting NWT_{next} is inspired by the two-bit branch predictor counter used in some high performance processors.

4.7.2 Distributing SystemC Objects for Load Balancing

The dynamic behavior of the simulated system means workloads of threads can gradually become imbalanced. This imbalance harms simulation performance. As discussed in Section 4.3, at the end of periods, the multithreaded engine checks the workloads of threads and tries to keep loads balanced among threads by redistributing SystemC objects. Thus, a metric that can be evaluated by the engine with low cost should be defined to indicate the seriousness of imbalance among workloads.

Assume that k ($k \geq 2$) worker threads are used in a period during a simulation, we define the *imbalance ratio (IR)* as $MAX(ABS(\frac{w_i - w_{avg}}{w_{avg}}))$ where w_i is the workload of thread i

($1 \leq i \leq k$) and w_{avg} is the average of workloads of threads. Functions MAX and ABS calculate the maximum value and the absolute value respectively. If IR is higher than a threshold value, namely *threshold imbalance ratio* (TIR), a serious imbalance happens and a redistribution (repartitioning) is triggered.

The multithreaded engine redistributes the SystemC objects into partitions as follows. The engine updates the weights of vertices of the partition graph for the simulated system with the computation of modules within the current period. The computation of a module within a period is the aggregate time for executing the invoked processes contained by this module in all the simulated cycles within the period. Then, the number of partitions is set to the number of worker threads for next period, i.e., NWT_{next} . The engine partitions the partition graph into NWT_{next} partitions using the technique described in Section 4.6.

4.7.3 Strategies for Adaptive Multithreaded Simulations

In this subsection, we present two strategies for adaptive multithreaded simulations. The major difference between these strategies lies in how the parameters needed for adaptive simulations are determined. Three parameters are needed by the multithreaded engine to carry out adaptive simulations: length of period (LOP), threshold imbalance ratio (TIR) and number of worker threads (NWT).

We also introduce two metrics for studying repartitioning. *Repartitioning cost* (RC) is the ratio of time spent on repartitioning to time spent on simulation in a period. *Repartitioning frequency* (RF) is the ratio of the number of periods in which repartitionings occur to the total number of periods in question.

4.7.3.1 Strategy for non-automated adaptive simulations

With this strategy, for a simulation, all the above three parameters are provided to the multithreaded engine by simulator developers. Additionally, they are kept the same from the beginning to the end of the simulation. Since NWT cannot be adjusted during simulation, the decision (4) and the process (5) of the flowchart in Figure 4.3 are skipped under this strategy. However, the decision (6) and the process (7) of the flowchart are kept, i.e., the engine checks workloads of threads and tries to achieve load balance.

The performance of simulation may not be optimal under this strategy due to the fact that parameters can not be adjusted. However, we use this strategy for the below purposes: 1)

We show in experiments the effectiveness of our framework and the necessity of adjusting parameters during simulations. 2) We study impacts of parameters on relevant metrics via experiments. 3) Study of non-automated adaptive simulations paves the way for the realization of fully automated adaptive simulations which is discussed later.

4.7.3.2 Strategy for fully automated adaptive simulations

Under this strategy, a simulation runs following the complete flowchart in Figure 4.3. Moreover, developers don't worry about above parameters, including NWT , TIR and LOP , which are determined automatically by the engine, at the process (2) of the flowchart in Figure 4.3. The details are explained as follows.

As discussed in subsection 4.7.1, NWT is initialized as the number of physical cores on the multicore host when a simulation starts. During simulation, NWT is adjusted by the engine following the variations of simulation computation.

In Section 4.8, several experiments of non-automated adaptive simulations are carried out to study the impacts of parameters TIR and LOP on the metrics RC and RF . After the relations between parameters and metrics, which are described in Section 4.8, are established, we can use runtime evaluations of metrics RF and RC to give feedbacks to the engine so as to adjust TIR and LOP when necessary. However, the runtime evaluation of metrics RF and RC are different from the definitions for the whole simulation, they are denoted as RF_{rt} and RC_{rt} respectively. We describe these adjustments as follows.

When a simulation starts, TIR takes a reasonable initial value that is chosen based on the study of impacts of TIR in experiments. Thus, TIR can be adjusted based on the runtime metric RF_{rt} in order that the metric can be kept in a predefined range $[RF_{min}, RF_{max}]$. If $RF_{rt} > RF_{max}$, i.e., there are too many repartitionings, TIR is increased by a certain value. Otherwise, if $RF_{rt} < RF_{min}$, TIR is decreased by a certain value.

To calculate RF_{rt} , a counter is added to record the number of repartitionings since TIR is set to a particular value. The counter increases by 1 when a repartitioning occurs. When TIR is set to a new value, the counter is reset to zero. After every m periods, RF_{rt} is calculated as $RF_{rt} = \frac{cnt}{m \times x}$, where cnt is the reading of the counter, m is called “feedback interval” and commonly taken as 100 and $x = 1, 2, 3, \dots$

Similarly, when a simulation starts, LOP takes a reasonable initial value which is chosen based on the study of impacts of LOP in experiments. Thus, LOP can be adjusted based on

the runtime metric RC_{rt} in order that the metric can be kept in a predefined range $[RC_{min}, RC_{max}]$. If $RC_{rt} > RC_{max}$, i.e., the repartitioning cost is relatively high in a period of LOP cycles, LOP is increased by a certain value. Otherwise, if $RC_{rt} < RC_{min}$, LOP is decreased by a certain value.

To calculate RC_{rt} in each period, two pairs of RDTSC instructions are added to source code. One pair of instructions are added at the beginning and the end of the code for repartitioning. They can evaluate the time spent on repartitioning. The other pair of instructions are added to capture the time used to simulate a period, similar to the method used to capture the time for simulating a cycle in Section 4.5.3. RC_{rt} is calculated as the ratio of the time captured by the first pair of instructions to the time captured by the second pair of instructions.

4.8 Experiments and Results

Our framework is based on dividing SystemC modules into partitions and completes simulations of partitions using threads. Thus, our framework is general for any simulator with any micro-architecture as long as it has multiple modules. Conclusions from experiments in this section can be generalized to different microarchitectures.

Recently, people are increasingly interested in NoC based homogeneous MPSoCs [TCR⁺09] [Til09]. Thereby, our chosen simulated systems are such MPSoCs connected by meshes whose general architecture is shown in Figure 4.8.

4.8.1 Experimental Setup

4.8.1.1 Simulation host

All our experiments are carried out on a multi-core computer under low load. This machine has two Intel Xeon Quad Core X5460 CPUs and provides a total of 8 cores for computation. All cores share 12M L2 cache. The size of its main memory is 42GB. The processor clock speed is 3.16GHz. The machine runs Red Hat Enterprise Linux 5.

During experiments, this 8-core multicore computer was the most powerful available and therefore the number of worker threads is limited to 8 in this chapter. However, since our framework is portable and scalable, higher speedups can be achieved when more powerful simulation hosts are available.

4.8.1.2 Architecture of the simulated manycore chip

Figure 4.8: An NoC-based homogeneous MPSoC with 4×4 mesh

Figure 4.8 shows the microarchitecture of the simulated multi-/many-core chip. Tiles of the chip are connected by a 2-D mesh whose size is configurable. Components of a tile are shown in Figure 4.8 (b). Several components including CPU, cache, local memory and network interface (NI) are connected to a local bus. The NI is also connected to the CPU as an accelerator to the CPU. The router is similar to that in [MWM04]. The NI acts as a bridge between the router and other components inside the tile. The simulator for the simulated multi-/many-core chip is developed based on UNISIM. More details of the simulator can be found in the preceding chapter.

4.8.1.3 Software for the simulated manycore chip

Several instructions such as “send” and “recv” are added to enable CPUs to communicate with each other via the network-on-chip. A GNU based cross-compiler tool chain is created for the compilation of these instructions. Additionally, a subset of MPI APIs is realized atop the above instructions. Moreover, a Linux OS plug-in is attached to each CPU to run statically linked ELF32 executables. This simulator can run message-passing parallel applications. We run a simple MPI based parallel program [PI09] where π is calculated for 64×1024 steps on the above simulator. The whole calculation is evenly distributed to all computing cores of the simulator.

4.8.2 Performance Evaluations

4.8.2.1 Baseline simulations

In experiments, several simulators are generated with different mesh sizes: 2×2 , 4×4 , 8×8 , 12×12 and 16×16 . To evaluate our framework, we use speedup, which is defined below, as the first metric. To provide baselines for comparison, simulations are firstly driven by the single-threaded UNISIM SystemC engine. Time for single-threaded simulation is denoted as T_s . Details of simulators and simulation results are shown in Table 4.1.

Table 4.1: Configurations for simulators and T_s

Mesh Size	Module #	Process #	Signal #	Simulated Cycles	T_s (seconds)
2×2	120	616	913	69,408,992	5,765
4×4	592	3,056	4,777	17,444,819	8,378
8×8	2,592	13,408	21,433	4,842,836	32,217
12×12	6,000	31,056	49,993	3,138,886	107,507
16×16	10,816	56,000	90,457	3,076,238	215,217

While the same calculation is run on the simulators, simulators with larger mesh sizes have more simulated CPU cores for computing and simulations on them thus take less simulated cycles to complete, shown in Figure 4.9 (a). However, as simulators with larger mesh sizes have more SystemC objects and thereby there is much more computation in each simulated cycle, simulations on them take longer time to finish on the simulation host computer, shown in Figure 4.9 (b).

4.8.2.2 Evaluation of non-automated adaptive simulations

Above simulations for different mesh sizes are then driven by the multithreaded engine. For this, we simply modify a parameter for execution mode in the configuration file of simulator and there is no need to recompile the simulator or the MPI program.

We first apply the non-automated adaptive simulation strategy in experiments. Parameters *LOP*, *TIR* and *NWT* are provided manually through the configuration files of simulators and passed to the multithreaded engine.

Speedup of a parallel simulation p is defined by $S_p = \frac{T_s}{T_p}$, where T_p is the time for running simulation p using the proposed multithreaded engine and T_s is time for single-threaded simulation.

4.9.a:

4.9.b:

Figure 4.9: Baseline simulations driven by the single-threaded UNISIM cycle-level engine

In this group of experiments, without losing generality, LOP , TIR and NWT are taken as 50, 5% and 8 respectively. Results are shown in Table 4.2 and also illustrated in Figure 4.10.

Table 4.2: Results of non-automated adaptive simulations

$LOP = 100$ cycles, $TIR = 5\%$, $NWT = 8$					
Mesh Size	T_p (seconds)	S_p	Mesh Size	T_p (seconds)	S_p
2×2	17,429	0.33	8×8	5,899	5.46
4×4	11,212	0.75	12×12	18,485	5.82
			16×16	43,248	4.98

Among simulations in Table 4.2, the two with smaller meshes are slowed down by using 8

Figure 4.10: Speedups in non-automated adaptive simulations

threads. Because the proportion of computation that can be made parallel in each simulated cycle is small, benefits from parallel simulations cannot overcome overheads. Using less worker threads could increase the proportion of parallel computation and reduce synchronization costs. Thus, it is necessary to adjust number of worker threads in simulation to follow varying simulation computation.

The other three simulations prove that the proposed framework does notably accelerate simulations. However, speedup obtained in simulation of simulator with 16×16 mesh is smaller than both speedups of those with meshes of size 8×8 and 12×12 respectively. The explanation is as follows. As shown in [PFH⁺06], the overall speedup benefits from the aggregate cache capacity and thread level parallelism. As these three simulations all run with 8 threads, we consider that they benefit from parallel execution speedup similarly. The simulators with 8×8 and 12×12 meshes boost their performance because their data can be fitted into the aggregate cache, which includes the *L1* and *L2* caches. However, the data of the simulator with 16×16 mesh are too large to be maintained in the aggregate cache. So, the performance of this simulator degrades because the number of cache misses increases. Using hosts with larger caches or leveraging intelligent cache replacement algorithms may improve this which however are out of the scope of this thesis.

4.8.2.3 Impacts of Threshold Imbalance Ratio (*TIR*)

We choose an accelerated simulation by the simulator with 12×12 mesh to evaluate impacts of *TIR* on repartitioning. Experiments have been done after *NWT* and *LOP* are taken as 8 and 100 respectively. Results are shown in Table 4.3 and illustrated in Figure 4.11.

Table 4.3: Impacts of Threshold Imbalance Ratio (TIR)

Mesh: 12×12 , $NWT = 8$, $LOP = 100$ cycles, TIR changes					
TIR	2.5%	5.0%	7.5%	10.0%	12.5%
T_p	18,509	18,485	18,242	18,076	18,539
S_p	5.81	5.82	5.89	5.95	5.80
RC	1.1570%	1.1452%	1.1617%	1.1469%	No repartitioning occurs
RF	99.9675%	32.4816%	3.1818%	0.0358%	0.0000%

4.11.a:

4.11.b:

Figure 4.11: Impacts of Threshold Imbalance Ratio (TIR)

Shown in Figure 4.11 (a), under different TIR s, RC s are very close because computation in a same period is same and costs of partitioning SystemC objects are close. But RF s under different TIR s have large differences. A smaller TIR causes more repartitionings, i.e., a higher

RF, but is more sensitive to workload imbalances. A larger TIR may miss chances of adjusting workloads promptly. Shown in Figure 4.11 (b), speedups are very close when TIR s are in the range shown in Table 4.3.

4.8.2.4 Impacts of Length Of Period (LOP)

We choose the same accelerated simulation by the simulator with 12×12 mesh to evaluate impacts of LOP on repartitioning. Experiments have been done after NWT and TIR are taken as 8 and 5% respectively. Results are shown in Table 4.4 and illustrated in Figure 4.12.

Table 4.4: Impacts of Length Of Period (LOP)

Mesh: 12×12 , $NWT = 8$, $TIR = 5.0\%$, LOP changes					
LOP (cycles)	20	50	70	100	150
T_p	18,821	18,645	18,281	18,485	18124
S_p	5.71	5.77	5.88	5.82	5.93
RC	5.7954%	2.3086%	1.6681%	1.1452%	0.7779%
RF	34.1049%	44.0672%	20.9725%	32.4816%	13.9214%

Shown in Figure 4.12 (a), RC decreases when LOP increases. It is because costs of partitioning the graph and distributing objects are same but amounts of computation in periods with longer LOP s are larger. Under different LOP s, the multithreaded engine has different checkpoints in these simulations and there is no point in comparing the RF s. Shown in Figure 4.12 (b), speedups are quite close when LOP s are in the range listed in Table 4.4.

Speedups in Table 4.3 and 4.4 show that LOP and TIR can be adjusted to manage load balance so as to improve speedup for a given simulation. However, the difference between speedups achieved under different values of LOP and TIR is not that significant. On the other hand, there are quite obvious relations between metric RF (RC) and parameter TIR (LOP). Therefore, we can use these relations to give feedbacks to the engine in order to adjust parameters in fully automated simulations as discussed in Section 4.7.3.2.

4.8.2.5 Evaluation of fully automated simulation

In this group of experiments, we apply the fully automated strategy in multithreaded adaptive simulations.

The predefined range for runtime Repartitioning Frequency (RF_{rt}) is set as $[30\%, 70\%]$, i.e., $RF_{min} = 30\%$ and $RF_{max} = 70\%$. During simulation, if RF_{rt} does not fall in the range, the

4.12.a:

4.12.b:

Figure 4.12: Impacts of Length Of Period (LOP)

multithreaded engine adjusts TIR accordingly. When modified, TIR is increased or decreased at a step of 2.5%.

The predefined range for runtime Repartitioning Cost (RC_{rt}) is set as $[2.5\%, 10\%]$. In simulation, if RC_{rt} falls outside of the range, the multithreaded engine adjusts LOP accordingly. When modified, LOP is increased or decreased at a step of 10.

When a simulation starts, default values for parameters are used: $NWT = 8$, $TIR = 7.5\%$ and $LOP = 50$. In simulation, whenever NWT is changed, modified values of TIR and LOP are discarded and they are restored to default values.

Simulations for different mesh sizes are driven by the multithreaded engine under the fully-

automated adaptive simulation strategy. Results are shown in Table 4.5 and illustrated in Figure 4.13. Column “Range of NWT ” indicates NWT decided by the engine to follow computation variations, excluding values that are taken when a simulation starts.

Table 4.5: Results of fully-automated adaptive simulations

Mesh Size	T_p (seconds)	S_p	Range of NWT
2×2	5,803	$0.99 \approx 1$	1
4×4	7,537	1.11	$4 \sim 7$
8×8	5,737	5.62	8
12×12	18,359	5.86	8
16×16	42,986	5.00	8

Figure 4.13: Speedups in fully automated adaptive simulations

Compared to performance in non-automated simulations using 8 worker threads, performance of fully automated simulations with mesh size 2×2 and 4×4 has been improved. Simulation with 2×2 mesh finishes after running for the time close to T_s . Simulation with 4×4 mesh even achieves a small speedup of 1.11. The main reason for improvements is the adjustment of NWT , shown in column “Range of NWT ” of Table 4.5, though there are adjustments of TIR and LOP in simulation. Suitable NWT are decided at runtime to match varying computation. Figure 4.14 shows values of NWT within 1,000 and 500,000 simulated cycles for these simulations.

Simulations for mesh size 8×8 , 12×12 and 16×16 have been accelerated. Their large amounts of computation during simulation can always support 8 worker threads to run efficiently, as shown in column “Range of NWT ” of Table 4.5. Though the known highest speedup

Figure 4.14: Number of worker threads in periods during fully automated simulations

of 5.95 has not been achieved in fully automated simulations, speedups achieved in them are close to those achieved in non-automated simulations by using 8 worker threads.

Based on above experiments under two strategies, *NWT* is the decisive one among parameters for determining speedups achieved in adaptive simulations. When other parameters have reasonable values, the fully automated adaptive simulations can achieve notable speedups comparable to those in non-automated simulations.

4.9 Summary

We have presented a systematic framework for accelerating UNISIM cycle-level microarchitectural simulations on multicore platforms. A unique approach that transforms the original single-threaded UNISIM engine into a multithreaded engine has made it possible to exploit the fine-grained parallelism within simulated cycles. We have shown that the SystemC objects can be divided into partitions so as to facilitate concurrent simulations. The adaptive technique for managing the overall computation variations and the workloads of threads has been shown to improve performance notably. A simulation can be fully automated by relying on runtime feedback to adjust the number of worker threads and the workloads of threads to improve performance. The proposed framework also allows for employing the non-automated adaptive

simulation strategy in order to allow users to manage adaptive simulations on a case-by-case basis.

Our experiments to simulate manycore implementations on an 8-core multicore computer show that speedups of close to 6X can be achieved. It is also shown that number of worker threads is the predominant parameter for achieving notable acceleration in adaptive multi-threaded simulations.

4.9.1 Novelty of Our Research

Our work on the acceleration of architectural simulations is different from the techniques mentioned in Section 2.3.3 in that our work owns the below features at the same time: 1) our technique works at the simulation engine level and all simulators based on the same engine can benefit from it; 2) our technique can be generalized to any discrete event simulation engine with delta-delay semantics; 3) our technique doesn't require modification to the source code of simulators such that the accuracy of simulators is not compromised; 4) our technique leverages parallel processing and dynamic load balancing techniques for high performance.

Chapter 5

A Scalable Strategy for Runtime Resource Management

5.1 Introduction and Motivation

The “manycore era” [ABC⁺06] [Bor07] is approaching, where processors with a large number of CPU cores (hence named “manycore” processors) will become ubiquitous not only in server and desktop machines but also in small client devices. A manycore processor may have hundreds, even thousands of small but energy-efficient CPU cores and uncores, which are likely encapsulated into tiles that are connected by an NoC. Such tile-based architectures overcome the limitations of wire delay, reduce design complexity and power consumption, and hence enable scalability of designs [BD02]. The most popular NoC topology is 2D mesh because it is regular, simple and predictably scalable with regard to power and area [BD06].

One category of manycore processors is NoC based homogeneous (all tiles are identical) manycore processors with 2D mesh topology for general-purpose embedded computing. This category of processors is referred to “*embedded manycore NoCs*” in this thesis. The embedded manycore NoCs are of interest to many researchers [CM07] [CM08] [COM08] [TCR⁺09] [CM10]. An example from industry is Tiler’s 100-core TILE-Gx100 [Til09]. It has been applied in embedded markets such as multimedia and networking.

Embedded manycore NoCs can execute several applications concurrently. Each application can have multiple tasks running in parallel, using multiple CPU cores. As these NoCs will be used in devices such as smart phones and PDAs, users can start/stop applications that lead to dynamic system configurations in term of the use of CPU cores. Such dynamic configurations are extremely difficult to model off-line. Hence, run-time techniques are indispensable.

Several run-time strategies [CM07] [CM08] [COM08] [CM10] have been proposed for resource management on embedded NoCs, which is event-driven. When an application *App* enters the system, suitable resources, i.e., CPU cores (a core is a basic unit to execute tasks), are identified and allocated to tasks of *App* (*allocation process*). When *App* finishes, the resources *App* occupies are reclaimed by the system (*deallocation process*). These strategies designate a core as the global manager (*GM*) that is responsible for resource allocation and deallocation when events occur. Moreover, to fully utilize resources, *GM* identifies and allocates resources in forms of irregular regions to applications. Fig. 5.2 shows how applications in Fig. 5.1 (described in Application Communication Graphs, i.e., ACGs [PdWSvM06]) are mapped to a 5×5 embedded NoCs. These applications are executed on five irregular areas which are highlighted and enclosed by lines in different colours. The tile at the left-top corner is assumed to be the *GM* and runs the OS. The other 24 tiles are controlled by *GM*.

Figure 5.1: The application characteristics of five applications

Figure 5.2: Mapping of five applications in Fig. 5.1 to a 5×5 NoC

However, the above strategies have difficulties when applied to embedded manycore NoCs with large core counts. The centralized resource management by *GM* could make *GM* the performance bottleneck of the system and impede the scalability as the core counts of manycore NoCs increase. As irregular regions are identified and allocated to applications, the applications suffer both *internal* and *external communication contentions* caused by messages from tasks belonging to the “same” or “different” applications contending for links, respectively.

To overcome above limitation of scalability due to the centralized resource management and communication contentions, we propose a scalable hierarchical strategy for runtime resource management on embedded manycore NoCs. First, our strategy uses submeshes in resource organization and accommodates applications with resources in forms of submeshes. A *sub-mesh* is defined as a rectangular area consisting of multiple cores which helps avoid external communication contentions.

Moreover, our strategy handles resource management in a hierarchical way. (1) A scalable scheme is adopted by the *GM* to manage submeshes: a submesh is identified and allocated to an incoming application and the submesh is reclaimed after the application finishes. (2) A CPU core on a submesh is chosen to be the *local manager (LM)* which manages the resources within the submesh. (3) In managing resources of a submesh, the *GM* communicates with its *LM* and the *LM* communicates other cores within the submesh. In this way, the communication costs including latency and energy consumption can be reduced compared to those under the centralized resource management of the previous strategies.

5.1.1 Contributions and Chapter Organization

The contributions of this chapter are as follows:

- Submesh is introduced and justified as the basic unit for resource management on embedded manycore NoCs in order to avoid external communication contentions among applications.
- A hierarchical resource management strategy is proposed to achieve a scalable resource management strategy where resource management is implemented at two levels.

The rest of this chapter is organized as follows. Section 5.2 introduces necessary background for the discussions in the following sections. Section 5.3 presents submesh based resource

management. Section 5.4 presents our proposed strategy for runtime resource management, including its hierarchical mechanism for resource management and the off-line preprocessing of the applications. Section 5.5 shows the experiments and the related results. Section 5.6 concludes this chapter.

5.2 Preliminaries

5.2.1 Overview of Existing Strategies

Figure 5.3: An NoC with 5×5 2D mesh used in existing strategies

In order to facilitate further discussions, we present the overview of the existing strategies in [CM07] [CM08] [COM08] [CM10]. As shown in Figure 5.3, the platform used in existing strategies consist of identical tiles and each tile consists of a CPU core, local memory and other components. This platform has two separate NoCs with 2D mesh topology: a data network and a control network. The data network transmits data messages needed for computation between tiles. The control network is used to pass control messages from other tiles back to a special tile (the tile at the top-left corner in Figure 5.3), i.e., the global manager (GM), which runs OS, for notifying it that the assigned tasks have been completed and they are ready to take other tasks. Data network is separated from control network to ensure that messages in the two networks do not interfere. Each tile has two routers (R_d and R_c) and two network interfaces (NI_d and NI_c).

The above existing strategies share the following features in resource management:

- *Centralized resource management.* Each tile can be seen as a relatively complete subsystem and is seen as a basic unit in resource management. One tile acts as the global manager (GM) and is responsible for resource management when system events occur.

- *Event-driven resource management.* When applications enter the system (one kind of event), *GM* identifies and provides computing resources to them. This process is called “allocation”. When applications complete their executions (the other kind of event), the resources allocated to these applications are reclaimed by *GM*. This process is called “deallocation”. The *GM* carries out the resource management when events occur.
- *Non-preemptive multi-tasking.* As non-preemptive multi-tasking is supported, tasks of an application are not reallocated or switched out from the CPU cores once they start executing on the cores allocated to them.
- *Irregular allocated regions for applications.* In order to fully utilize the computing resources, the global manager can identify and allocate irregular regions to applications.

Centralized resource management is realized by the *GM*. The *GM* runs the operating system which is assumed to be a compact and efficient real-time OS and further supports event-driven programming. *GM* continuously tracks the status of the tiles/CPU cores (idle or used) and is responsible for system resource management when system events occur.

5.2.2 Model of Applications

The model of applications is introduced here. Similar to [PdWSvM06], the applications are described by the Application Communication Graph $ACG = (V, E)$ and represented as *directed* graphs with the following properties.

- (i) *Vertices.* Each vertex v_i in V represents a cluster of tasks. Tasks belonging to the same cluster should run on a same tile. Additionally, each vertex v_i has its minimum computation requirement at which it should operate in order to meet the application deadlines.
- (ii) *Edges.* Each directed edge e_{ij} in E characterizes the communication from vertex v_i to vertex v_j , while weight $w(e_{ij})$ stands for the communication rate, (i.e., bits per time unit) from vertex v_i to vertex v_j .

As shown in Figure 5.1, five applications have the application characteristics, i.e., the Application Communication Graphs (*ACGs*). The model of application *App1*, i.e., ACG_{App1} , has three vertices and three edges. The weight of each edge of ACG_{App1} is 2.

5.3 Submesh-based Resource Management

5.3.1 Submeshes for Resource Management

As embedded manycore NoCs are similar to large-scale parallel and distributed systems where submesh is used in resource management, submesh is introduced into for resource management on embedded manycore NoCs. A *submesh* is defined as a contiguous and rectangular area within the whole mesh of the NoC. It may contain several tiles and thereby multiple CPU cores. Using submeshes as basic units in resource management on manycore NoCs could lead to several benefits as follows.

First, a submesh provides a natural and coarser granularity than a single core for assigning resources to applications. As the CPU cores of manycore systems are small and simple in terms of architectural features [ABC⁺06], the performance of an individual core is not as high as the cores in current aggressive multicore processors. Hence, it is necessary that applications are parallelized and multiple CPU cores cooperate to carry out computation so as to provide reasonable performance. In addition, as the number of cores in manycore processors is large, using submeshes helps reduce the complexity in resource management.

Second, when applications are mapped onto submeshes, they can avoid the external communication contentions. It has been shown in [CM10] that tasks of the same application don't suffer from external communication contentions when these tasks run on a convex region. By definition, a submesh is a rectangular contiguous region that is obviously convex. So, when applications are executed on their respective submeshes, there is no external communication contention among these applications. In addition, a submesh is a contiguous area where cores are in close proximity which keeps the costs of internal communication low.

Third, submeshes help achieve performance isolation on manycore NoCs. It has been suggested in [ACJ⁺] that several partitions should be supported on a tera-scale processor where each partition has a fraction of the total number of various platform elements. Moreover, it is desirable that the performance of each of these multiple partitions is not affected by the performance of other partitions so as to satisfy Quality of Service (QoS) requirements. When applications are executed on their respective submeshes, traffic generated in one submesh does not interfere with traffic from another submesh which helps keep QoS of applications.

We introduce several notations for submeshes with the help of Figure 5.4 where five applications have been mapped to and are running on the 64-tile platform with an 8×8 mesh. These

applications App_i are executed on areas S_i , $1 \leq i \leq 5$, respectively. The areas S_i , $1 \leq i \leq 5$, are submeshes with dimensions ($width \times height$): 2×2 , 3×2 , 3×2 , 3×2 and 3×2 , respectively.

First, an embedded manycore NoC with 2D mesh is denoted as $M(W, H)$, where W is the width of the mesh and H is its length. The NoC in Figure 5.4 is denoted as $M(8, 8)$.

A tile located at the i th row and the j th column is denoted as $tile_{i,j}$. Note that: an internal tile is directly connected to four neighbors; the four mesh corner tiles have two neighbors each; and each remaining boundary tile has three neighbors.

Symbol $PS_{i,j}^{r \times s}$ is used to represent a physical submesh with $tile_{i,j}$ as its left-top corner and $r(s)$ is the number of rows (columns) of the submesh. For example, in Figure 5.4, submesh S_1 is annotated as $PS_{1,1}^{2 \times 2}$ and submesh S_4 is annotated as $PS_{6,3}^{3 \times 2}$.

Figure 5.4: Submeshes on a 64-tile manycore NoC

5.3.2 Algorithms for Submesh Allocation and Deallocation

After submesh is introduced, we consider the algorithms for allocation and deallocation processes. Submesh-based processor allocation schemes have been successfully applied to multi-computer systems with mesh topology and many good algorithms, such as [LC91, Zhu92, DB93, CT94, LHLB95, SP96, KY98, CC99, Aba06, LWLN97, BMOKAM07, BMOKA07], have been proposed for allocation and deallocation in the literature. These schemes have been widely used in parallel and distributed systems for performance optimization and resource management. And they aim to maximize the utilization of processors and system performance.

The free-list submesh allocation scheme proposed in [Aba06] is adopted by us because it is better than other contiguous schemes in terms of scalability and algorithmic complexity.

This scheme maintains an unordered list of possibly overlapped free submeshes. A tile is *free* if it is not allocated to any application, and a *free submesh* is a physical submesh all of whose tiles are free. When all cores are not occupied, the list contains only a free submesh, i.e., the whole mesh.

For allocation, it selects the first free submesh that has at least the same size as the request, and when the selected submesh is larger than the request the part actually allocated is one that has the largest boundary value. The *boundary value of a free submesh* is defined as the sum of the boundary values of the tiles located on its periphery, where *the boundary value of a tile* is the sum of the number of allocated neighbor tiles and the number of mesh edges on which the tile lies. Allocating free submeshes with largest boundary values leads to *compaction*, i.e., higher resource efficiency.

After successful allocation and deallocation of applications, the scheme dynamically splits or combines submeshes to make sure that only *maximal submeshes* (A submesh is a maximal free submesh if it is free and not contained in any other free submesh) are kept in the list.

The time complexity of the scheme is linear in the number of free submeshes which is better than those of the best previously proposed schemes having time complexities that are either quadratic or cubic in the number of free or allocated submeshes.

Here, we show how the allocation and deallocation algorithms work through the example in Figure 5.4. Five applications App_i enter the system in sequence which require resources in submesh dimensions 2×2 , 3×2 , 3×2 , 3×2 and 3×2 , respectively, where $1 \leq i \leq 5$. The system configurations and the content of $LIST$ at different allocation/deallocation stages are shown in Figure 5.5.

At the beginning there is no application running on the system and thereby all tiles of the NoC are free. The list for free submeshes, i.e., $LIST$, has only one element $PS_{1,1}^{8 \times 8}$. Therefore, $LIST = \{PS_{1,1}^{8 \times 8}\}$ and the system configuration is shown in Figure 5.5.a.

Then, application App_1 , which requires a 2×2 submesh, comes to the system. OS executes the allocation algorithm and submesh S_1 is identified for App_1 . After S_1 is allocated, $LIST = \{PS_{1,3}^{8 \times 6}, PS_{3,1}^{6 \times 8}\}$. The system configuration is shown in Figure 5.5.b.

Figure 5.5: System configurations at allocation/deallocation stages

After that, application App_2 , which requires a 3×2 submesh, comes to the system. OS identifies submesh S_2 for App_2 . After S_2 is allocated, $LIST = \{PS_{1,3}^{8 \times 6}, PS_{6,1}^{3 \times 8}\}$. The system configuration is shown in Figure 5.5.c.

Similarly, after S_3 is allocated for App_3 , $LIST = \{PS_{1,3}^{8 \times 6}\}$. The system configuration is shown in Figure 5.5.d. After S_4 is allocated for App_4 , $LIST = \{PS_{1,3}^{5 \times 6}, PS_{1,5}^{8 \times 4}\}$. The system configuration is shown in Figure 5.5.e. After S_5 is allocated for App_5 , $LIST = \{PS_{1,3}^{5 \times 6}, PS_{1,7}^{8 \times 2}\}$. The system configuration is shown in Figure 5.5.f. Of note, as the list of free submeshes is unordered, there could be several different system configurations after the above 5 submeshes have been allocated.

If there are no other applications coming to the system and the application running on S_4 finishes, the submesh S_4 is reclaimed by the system. After that deallocation, there are three free submeshes in the list, $LIST = \{PS_{1,3}^{5 \times 6}, PS_{1,7}^{8 \times 2}, PS_{1,3}^{8 \times 2}\}$. The system configuration is shown in Figure 5.5.g.

As shown above, the scheme maintains the free list to keep the maximal submeshes in it via splitting and/or combining free submeshes after allocation and deallocation processes.

5.4 The Proposed Hierarchical Strategy

5.4.1 Overview

In the existing strategies, the functionalities of resource management, which consist of 1) identifying resources for an application, 2) mapping its tasks to the identified resources, and 3) reclaiming resources allocated to the application when it completes, are executed on a single tile, i.e., the GM alone. This could make the GM become the system performance bottleneck when the number of the CPU cores on the embedded manycore NoCs increases.

In our proposed hierarchical strategy, after submesh is introduced, the resource management is carried out by the GM and local managers in a hierarchical way. In other words, the functionalities of the resource management are divided between GM and local managers. The GM manages resources for applications in forms of submeshes. The resource management within submeshes is off-loaded from the GM to the local managers so as to reduce the GM 's chance of becoming the performance bottleneck.

For an example application App , the processes that it goes through under the hierarchical strategy are elaborated as follows.

- (i) When ACG of App is given, the off-line preprocessing of App decides the submesh dimension for App and how to map tasks of App onto the submesh.
- (ii) When App enters the system, the GM allocates a submesh S to App based on the required submesh dimension. The CPU core at the top-left corner of S is chosen as the local manager of S , denoted as LM_S .
- (iii) After S is allocated to App , the LM_S maps the tasks of App onto the cores on S .
- (iv) When a task of App completes, the core running this task sends a message to the LM_S to notify the task's completion. When all tasks of App complete, the LM_S sends a message to the GM to start a deallocation process.

Thus, the components of the proposed hierarchical strategy consist of preprocessing of applications, submesh management, and hierarchical resource management, which are discussed in following sections.

5.4.2 Off-line Preprocessing of Applications

In this section, we discuss the off-line preprocessing of applications. For any application App whose application characteristics are given, the purpose of the preprocessing is to determine: 1) dimensions of the submeshes that can be used to accommodate App ; 2) the mapping of App 's tasks onto the corresponding submesh when a dimension is chosen.

Note that there could be several suitable dimensions for the submeshes that can be used to accommodate App and for each dimension there is a mapping of App 's tasks onto the submesh. In this chapter, we present one method for preprocessing application. More methods for preprocessing can be investigated in future work.

After assuming that the applications are described by $ACGs$, i.e., application communication graphs, we adopt a model of communication energy consumption using the bit energy metric which is proposed in [YMB02]. With this model, the total communication energy consumption of any application App per time unit can be calculated using the following equation:

$$E_{App} = \sum_{\forall e_{ij} \in E \text{ in } App} w(e_{ij}) \times E_{bit}(e_{ij})$$

where $w(e_{ij})$ is the communication rate of an edge in application App (in bits per time unit), and $E_{bit}(e_{ij})$ stands for the energy consumption to send one bit from the tiles where vertices v_i and v_j are allocated to (in Joules per bit). More precisely,

$$E_{bit}(e_{ij}) = (MD(e_{ij}) + 1) \times E_{R_{bit}} + MD(e_{ij}) \times E_{link}$$

The parameter $E_{R_{bit}}$ stands for the energy consumed in routers, including the crossbar switch and buffers, where E_{link} represents the energy consumed in one unit link, for one bit of data; these parameters are assumed to be constant. The term $MD(e_{ij})$ represents the Manhattan distance between the tiles where vertices v_i and v_j are allocated to. Assume that the (X, Y) coordinates of v_i and v_j are (x_i, y_i) and (x_j, y_j) , then $MD(e_{ij}) = |x_i - x_j| + |y_i - y_j|$.

5.4.2.1 A Method for Preprocessing Applications

As discussed above, in our proposed submesh based strategy for runtime resource management, each application has its own submesh and the external communication contentions among applications are avoided. Therefore, each application can be considered independently in term of contentions and the contentions, which an application confronts, are the internal communication contentions caused by its own tasks.

When the dimension of the submesh for an application and the mapping of its tasks onto the tiles of the submesh are to be decided, our goal is to minimize the energy consumption by the application per time unit. As the dimension of a submesh has direct influence on the mapping of the tasks, we choose to consider these two factors together.

We propose the following method for preprocessing applications:

- (i) *Step 1.* To decide the dimension of a submesh for an application.
- (ii) *Step 2.* To decide the mapping of the application's tasks onto the submesh identified in Step 1.

In Step 1, we calculate the dimension of a submesh for an application based on the number of tasks inside an application, denoted as n . We search for a submesh $PS^{w \times h}$ whose number of tiles is equal or larger than n and the difference between width (w) and height (h) of the submesh is minimal. Such a submesh has enough resources, possibly with extra tiles which are not used/idle, to run the application. On the other hand, the average number of communication hops is minimized [BBD⁺08].

In Step 2, after the dimension of the submesh for the application is determined, the tasks of the application are distributed to tiles of the submesh where a task is allocated to a tile. We check all the possible distributions to find a distribution that makes the total communication energy consumption of the application minimal.

In the above steps, this repeats an exhaustive search on the dimension of the submesh and mapping of tasks onto the submesh. This is rational based on the following reasons. 1) The number of tasks of applications usually is small. Commonly, an application from the E3S benchmarks [Dic10] for embedded systems has less than a dozen tasks. 2) The above exhaustive search is carried out off-line by desktop machines. It does not take much time for desktop machines to complete the exhaustive search. For large scale problems, other algorithms need to be explored and we leave these for future work.

Such submeshes may have resources which are not used and are called “internal fragmentation”. According to the “utilization wall”, this internal fragmentation is not necessarily adversary. On future manycore NoCs where the numbers of cores on a single die are huge, idle cores can always be put into sleep to save power. For some systems, computing tasks could be migrated between idle cores and busy cores to keep the temperatures of cores balanced [BABP06].

For the five applications, App_i , $1 \leq i \leq 5$, whose application characteristics are shown in Figure 5.1, after the off-line preprocessing, the dimensions of submeshes are decided as follows: 2×2 , 3×2 , 3×2 , 3×2 and 3×2 , respectively. The mapping of tasks inside their allocated submeshes are shown in Figure 5.6, respectively.

Figure 5.6: Mapping of tasks of applications inside submeshes

5.4.3 Hierarchical Resource Allocation

Under the proposed hierarchical strategy, when an application requests for entering the system, the global manager first identifies a suitable submesh for it. After a submesh is allocated to the application, the local manager within the allocated submesh will map the tasks of the application onto the tiles of its submesh. We elaborate on this with the help of an example in Figure 5.7.

Figure 5.7: A hierarchical mapping of five applications to an 8×8 NoC

Figure 5.7 shows an example of a hierarchical resource allocation. In this example, the global manager, i.e., $tile_{1,1}$, executes the allocation algorithm to decide the five submeshes for applications App_i , $1 \leq i \leq 5$. The tiles at the left-top corners of submeshes are chosen as “local managers”. Usually, the tile that has the minimal distance to the global manager within the allocated submesh is chosen as the local manager. Each local manager maps the tasks of an application to the tiles within its submesh. For instance, the local manager of the submesh $PS_{3,1}^{3 \times 2}$, i.e., $tile_{3,1}$, to which the application App_2 is allocated, maps the tasks of App_2 to the five tiles of the submesh. Note that the local manager of the submesh $PS_{1,1}^{2 \times 2}$, which is allocated to App_1 , is exactly the global manager.

To support this hierarchical resource allocation carried out by the global manager and a local manager, related data structures at the global manager are designed as follows.

- The global manager maintains the list of free submeshes, as described above. The global manager also keeps a list of the allocated submeshes.

- When a submesh is allocated for an application, the tile at the left-top corner is designated as the local manager of the submesh. The data about the local manager are added and stored together with the data of the allocated submesh at the global manager.

On the other hand, a local manager has some supporting data structures as well. It is assumed that a local manager communicates with the global manager to get the following data from the global manager which are maintained by the local manager in its local memory.

- Information needed for a local manager to establish connection with the global manager.
- Information of both its submesh and tasks of the application to be run on the submesh.

5.4.4 Hierarchical Resource Deallocation

Within the existing strategies, when the processing of a task allocated on a tile is finished, the tile transmits control messages to the global manager via the control network in order to notify it that the current tile is ready for another task. Since the number of the tiles on embedded manycore NoCs can be huge, the global manager could become the performance bottleneck when completion of tasks occur. Therefore, another of our motivations is to reduce the number of tiles the global manager should communicate when these completion events occur.

We propose to realize a hierarchical resource deallocation process using the above mentioned local managers. In detail, a local manager is responsible for the control messages from the tiles inside its submesh. The other tiles inside a submesh don't send control messages directly to the global manager. Instead, they send control messages to the local manager of the submesh. When all tiles inside of a submesh have finished the processing of their tasks, the local manager notifies the global manager that the application has been completed and the submesh is ready for deallocation. In this way, most of the communication during resource deallocation is kept as local to submeshes.

In the above example in Figure 5.7, the local manager of submesh $PS_{3,1}^{3 \times 2}$, i.e., $tile_{3,1}$, accepts the control messages of other tiles within the submesh, i.e., $tile_{4,1}$, $tile_{5,1}$, $tile_{4,2}$ and $tile_{5,2}$. After the processing of all tasks on these tiles is completed, $tile_{3,1}$ sends a control message to the global manager to notify that the submesh $PS_{3,1}^{3 \times 2}$ is ready for deallocation.

The local managers are designed to work in a way that the need for them to manage their submeshes does not impact their computation efficiency for computation tasks. First, the computation tasks on a local manager only get started to run after the resource allocation within its

submesh has completed. In addition, a local manager only carries out the resource management of its submesh until the computation tasks running on the local manager have finished. Then, the local manager checks and de-allocates the resources whose tasks have completed. Furthermore, the local manager waits for the resources whose tasks have not completed to finish the whole resource deallocation process.

The proposed hierarchical resource deallocation process leads to the following benefits. First, the communication of the global manager can be greatly decreased as the number of tiles, with which it needs to communicate, is reduced. The energy consumed by the transition of control messages is also greatly reduced as most of the control messages are communicated locally, i.e., within the submeshes.

In addition, the local manager can implement certain application level optimizations. As the optimizations for various applications can be significantly different, they can be better realized with a local manager which knows its current application well.

5.4.5 An Illustrative Example

Figure 5.8: Mapping applications onto rectangular regions

Under the proposed hierarchical strategy, the five applications, which have been mapped to a 5×5 NoC as shown in Figure 5.2, can be mapped to a 5×6 NoC as shown in Figure 5.8.

In this example, when resource management events occur, the number of cores that the *GM* has to handle is greatly decreased, from 24 to 4. On the other hand, as submeshes are rectangular

areas, deciding on such regular areas is simpler than irregular ones. It is possible to decrease the time spent on executing the allocation algorithm by the *GM*. In this way, the chance of the *GM* to become the performance bottleneck is reduced.

As all applications are executed on the convex regions, they no longer suffer from external contentions. For the example shown in Figure 5.8, after one extra row of tiles, i.e., 5 tiles, have been provided for these applications, five rectangular regions, which are obviously convex, have been allocated to run the applications.

However, all applications are mapped onto regular regions under the condition that more resources have been provided to these applications and some resources are allocated but not used by tasks. For example, in Figure 5.8, for applications *App*₁, *App*₂ and *App*₅, one of the allocated tiles is not used. The possible idle resources within submeshes will not be an issue for embedded manycore NoCs because there will be a large amount of resources on future embedded manycore NoCs and not all of them be powered on by the system simultaneously according to the “utilization wall”.

5.5 Evaluation of The Proposed Strategy

We carry out two experiments to compare the existing resource management strategies, which are represented by strategy in [CM10], and our proposed hierarchical strategy. We focus on the comparisons of 1) the time spent on allocation process by the global manager and 2) the communication energy consumed during the deallocation process.

5.5.1 Experimental Setup

We first introduce the experimental setup which includes the target platform, the simulator, the bit energy metric model and simulation host.

5.5.1.1 Target Platform

The target platform is an embedded manycore NoC as shown in Figure 5.9, which has the same architecture with the embedded multiprocessor NoC in Figure 5.3. It has 30 tiles that are connected as an 5×6 2D mesh. As the size of the control network is dependent on the mesh size of the NoC, for this 5×6 NoC, $\lceil \log_2(5 \times 6) \rceil = 5$ bits are needed to decode the addresses of tiles. There is another bit to indicate the busy/idle status of the tile. The width of the links of the data network is assumed to 32 bits, i.e., the flit size is 32 bits.

Figure 5.9: An embedded manycore NoC with 5×6 2D mesh

5.5.1.2 The Simulator

Figure 5.10: The microarchitecture of a tile

We generate a cycle-accurate simulator for the above embedded manycore NoC, which is extended from the one described in Chapter 3. The MPI library and cross-compiler tool-chain presented in Chapter 3 are used here as well. The microarchitecture of each tile is shown in Figure 5.10.

Similar to the data network, a pair of instructions are added to the PowerPC ISA in order to enable the PowerPC 405 core to access the network interface (NI_c) of the control network. These instructions are only executed when the interruption point is met. The PowerPC 405 core can read/send control messages from/to the network interface NI_c .

5.5.1.3 The Bit Energy Metric Model

The bit energy metric model proposed in [YMB02] is adopted in the above simulator to evaluate the communication energy consumption in experiments where E_{link} is set to $4.49 \times 10^{-13} (J/bit)$ and $E_{R_{bit}}$ contains the energy consumes by the routing engine ($10^{-13} J/packet$), arbiter request ($1.155 \times 10^{-12} J/packet$), switch fabric ($2.84 \times 10^{-13} J/bit$), and buffer reading and writing ($1.056 \times 10^{-12} J/bit$ and $2.831 \times 10^{-12} J/bit$, respectively).

5.5.1.4 Simulation Host

The experiments are carried out on a multi-core computer under low load. This machine has two Intel Xeon Quad Core X5460 CPUs with a clock speed of 3.16GHz and providing a total of 8 cores for computation. All cores share 12M L2 cache. The size of its main memory is 42GB. This machine runs Red Hat Enterprise Linux 5.

5.5.2 Experimental Results

We apply our proposed hierarchical strategy to five real applications from the embedded system benchmark suite (E3S) [Dic10]: Automotive/Industrial, Consumer, Networking, Office automation, and Telecom. These same applications have been used in the experiments in [CM10]. These five benchmarks have been partitioned off-line using a method similar to [PdWSvM06].

The number of vertices in the ACG of each benchmark ranges from 3 to 6. In other words, the number of tasks of the benchmark applications ranges from 3 to 6. Their application characteristics are shown in Figure 5.1. We have shown results of the preprocessing in Figure 5.6.

We assume that these applications invoke the events, which are shown in Table 5.1, when they enter the system and leave the system. The global manager executes the allocation algorithm to allocate resources for an application when it enters the system. The global manager reclaims the resources of an application when the application finishes. Accordingly, the system configuration changes after an event occurs.

In both experiments, the events occur as described in the above table. However, the global manager carries out different allocation and deallocation algorithms in these different experiments. In the first experiment, the allocation/deallocation algorithms executed by the global manager are the ones presented in strategy of [CM10] focusing on minimizing the internal contention. In the second experiment, our proposed strategy for resource management is adopted

Table 5.1: Events in experiment

Event	Start Time	Incoming application	End Time
1	0	App_1	5
2	1	App_2	6
3	2	App_3	12
4	3	App_4	13
5	4	App_5	9

and hierarchical resource management is realized. These algorithms are implemented in C and are compiled using the above cross-compiler such that the algorithms can be executed on the PowerPC 405 CPU core.

5.5.2.1 Time of Executing Allocation Algorithms

As we are interested in the time spent executing the allocation algorithms by the global manager when events occur, we invoke the algorithms with different system configurations and application characteristics as inputs. The cycles used to execute the algorithms during simulations under the above events are listed in the Table 5.2. At time 4, after resources have been allocated to the application App_5 , two different system configurations following the different resource management strategies are shown in Figure 5.11. Note that the bottom row of Figure 5.11 (a) is not used in the allocation process.

Table 5.2: Execution cycles of allocation algorithms by the Global Manager

Event	1	2	3	4	5
Algorithm in [CM10]	122,776	892,368	1,092,368	1,322,875	1,262,536
Our Algorithm	121,890	281,337	132,047	301,752	298,266

We can see from the above table that the execution of allocation algorithm in [CM10] by the global manager generally takes more time than our proposed allocation algorithm. This is due to the following reasons. The allocation algorithm in [CM10] implements functionalities including identifying resources for an application and mapping tasks of the application to the identified resources. However, our proposed allocation algorithm only identifies the desired submesh. Additionally, complexity of our proposed algorithm is proportional to the number of the free submeshes in the list $LIST$.

5.11.a: Task allocation scheme under the strategy in [CM10] focusing on minimizing the internal contention

5.11.b: Task allocation scheme under our proposed strategy

Figure 5.11: System Configurations at Time 4

5.5.2.2 Communication Energy Consumption

We compare these two configurations in terms of communication energy consumption during deallocation process as follows. Communication energy consumption is calculated using the aforementioned bit energy metric.

The configuration in Figure 5.11 (a) spends energy consumption $7.92 \times 10^{-10} J$ for transmitting control messages when all tasks finish their execution. However, the configuration in Figure 5.11 (b) just spends $2.93 \times 10^{-10} J$. The reduction ratio is 63%.

The reason causing this great reduction is that our proposed strategy reduces the number of slaves that need to send control messages to the global manager from 24 down to 4. Therefore, the energy used to transmit the control messages to the global manager is greatly reduced. Most slaves just need to transit the control messages to their local managers which are much shorter in distances and the traffics are confined inside the submeshes allocated.

5.5.2.3 Internal Fragmentation

The costs for above saved time in allocation algorithm and saved energy in transmitting control messages are that some tiles allocated in some submeshes are not used in the execution of applications. They can be called “internal fragmentation”. For submeshes allocated to the applications App_1 , App_2 and App_5 , the “internal fragmentation ratio” are 25%, 16.7% and

16.7%, respectively. According to the “utilization wall”, this is not necessarily a bad thing in future manycore systems.

5.6 Conclusion

In this chapter, we present a scalable strategy for run-time resource management on embedded manycore NoCs which overcomes limitations of existing resource management strategies. Our proposed strategy accommodates applications with resources in forms of submeshes. A hierarchical resource allocation/deallocation mechanism is proposed in the strategy in order to increase the scalability of our proposed strategy.

Our experiments show that our proposed hierarchical strategy for resource management can reduce the time required for executing the allocation process and the energy used in transiting control messages during the deallocation process. These reductions make our proposed strategy more scalable than existing strategies. The above benefits are at the costs of leaving certain extra resources idle. However, according to the “utilization wall”, this is not necessarily adversary in future manycore systems.

5.6.1 Novelty of Our Research

To the best of our knowledge, our work on submesh based runtime resource management is the first one which introduces the submesh into the single-chip manycore systems based on the “utilization wall” on the future devices so as to avoid the external communication contentions among applications. In addition, the resource management is carried out in a hierarchical and scalable way which is different from centralized resource management adopted by the existing strategies.

Chapter 6

Hybrid Non-Preemptive/Cooperative Multi-tasking

6.1 Introduction and Motivation

As already discussed in preceding chapters, “manycore” processors will become the mainstream and run-time techniques are indispensable for resource management due to the dynamic system configurations at runtime.

Several run-time strategies [CM07] [CM08] [COM08] [CM10] have been proposed for task allocation. Their essence is to provide suitable resources to tasks of an application when the application enters the system. Resources are reclaimed after the application finishes. The existing strategies adopt centralized resource management and accommodate irregular regions of CPU cores to applications and thereby they could have limitation of scalability when the number of CPU cores of embedded manycore NoCs increases. In the preceding chapter, we have proposed a scalable hierarchical strategy to overcome the aforesaid limitation.

In addition, *the non-preemptive multi-tasking* has been adopted by the existing strategies, where tasks of an application are not switched out from the CPU cores allocated to them, once they start executing on these cores, until the application completes. Though the non-preemptive multi-tasking is simple, it may fail to launch new applications under certain circumstances. The below example of a 64-tile embedded manycore NoC illustrates such a case.

In Fig. 6.1, all cores are occupied. There are 5 applications, App_i , running on the areas S_i respectively, where $i = 1, 2, 3, 4, 5$. Now, the user tries to start an application J , which is assumed to request a area of 3×3 (9 tiles). Due to the non-preemptive multi-tasking, no core can be offered to J before some application completes. In another scenario, after App_i ,

$i = 1, 2, 3, 4$, complete and the areas they occupied are released, but App_5 is running on area S_5 , J still cannot get started because a 3×3 area cannot be found for J . Though there are idle CPU cores, J has to wait until App_5 is completed. This leads to low resource usage efficiency.

Figure 6.1: Concurrent applications on a 64-tile embedded manycore NoC

To the best of our knowledge, Tessellation manycore OS [LKB⁺09] [CBC⁺10] is a promising solution that has been proposed so far to improve the resource usage on manycore systems in above scenario. Particularly, Tessellation takes an aggressive approach of preemptive multi-tasking in resource management. However, it is still at its early research stage.

In this chapter, we propose a hybrid multi-tasking technique to overcome the limitations of the non-preemptive multi-tasking. We propose to introduce *cooperative multi-tasking for systems with multiple CPU cores* to embedded manycore NoCs. A *cooperative application*, which supports cooperative multi-tasking, can cooperate with the OS at runtime for 1) giving up some of its computing resources to help an incoming application get started; 2) obtaining extra resources so as to achieve higher performance if possible. For embedded manycore NoCs, it is not required for all applications to be cooperative. The non-preemptive and cooperative applications can co-exist and thereby we further propose a *hybrid non-preemptive/cooperative multi-tasking* to enable interactions among applications so as to reduce the chance of failure in launching applications and improve the resource usage efficiency.

6.1.1 Contributions and Chapter Organization

Contributions of this chapter are as follows:

- the cooperative multi-tasking for systems with multiple CPU cores has been introduced;
- a hybrid non-preemptive/cooperative multi-tasking technique is proposed to allow an application to increase/decrease its resources during the execution;
- the hybrid multi-tasking technique has been implemented and evaluated with a case study.

This chapter is organized as follows. Section 6.2 analyses various multi-tasking approaches and introduces the cooperative multi-tasking for systems with multiple CPUs. Section 6.3 presents the technique for hybrid non-preemptive/cooperative multi-tasking, including the architectural modifications and a method of parallelizing applications for the cooperative multi-tasking. In Section 6.4, an MPEG-2 encoder program is parallelized into a cooperative application as a case study and related experiments are carried out for evaluating the proposed hybrid multi-tasking. Section 6.5 concludes this chapter.

6.2 Cooperative Multi-tasking for Systems with Multiple CPUs

In this section, we review alternatives to non-preemptive multi-tasking, i.e., preemptive multi-tasking and cooperative multi-tasking, used in systems with a single CPU or multiple CPUs. Then, we introduce the cooperative multi-tasking for systems with multiple CPUs.

6.2.1 Multi-tasking Approaches for Single-CPU Systems

For preemptive multi-tasking implemented in systems with a single CPU, a computing resource, i.e., the CPU, is shared by several processes. These processes are handled based on time slices allotted by the OS. Essentially, each process is allotted a certain amount of processing time. When a time slice expires, execution of a process is interrupted so that another process can be run. In time-slice preemptive multi-tasking, whenever running of a process is interrupted, a variety of temporary values must typically be stored until this process resumes its execution. Preferably, the values are stored in local physical memory for fast context switching. Thus, time-slice preemptive multi-tasking is time and resource intensive because of many memory reads and writes and the need for memory for storing additional temporary values.

The other alternative to non-preemptive multi-tasking is cooperative multi-tasking. For cooperative multi-tasking implemented in systems with a single CPU, applications are designed with interruption points. When an interruption point of a process is reached, the OS is permitted to switch to another process. To save memory for temporary storage, designers can set interruption points where relatively few temporary values need to be stored.

However, for cooperative multi-tasking realized in systems with a single CPU, each application must be designed as a cooperative application. Otherwise, because the OS has no way to preempt operation, once an application begins execution, if it does not freely give up control, it will continue to execute until it terminates. This situation leads to no multi-tasking at all. On the other hand, the effects of designing all applications as cooperative ones by inserting desired interruption points are paramount. The above two problems could be the reasons that make cooperative multi-tasking unpopular in systems with a single CPU.

6.2.2 Preemptive Multi-tasking for Systems with Multiple CPUs

If the above preemptive multi-tasking is applied in embedded manycore NoCs, the time taken and memory required for storing temporary values required by multiple CPUs can be too huge to be afforded. Therefore, the above preemptive multi-tasking is very costly for embedded manycore NoCs whose valuable on-chip memory is limited.

In addition, when preemptive multi-tasking is applied to embedded manycore NoCs, there are extra concerns to be considered. Otherwise, this multi-tasking could lead to resource wastage under certain situations.

We explain this scenario with the help of the example in Figure 6.1. When application J comes, it requests a 3×3 submesh to get started. Based on the current system configuration, there are three possibilities if preemptive multi-tasking is applied.

- (i) If the application run on submesh S_1 is preempted and swapped out, though the number of resources on S_1 is large enough, the shape of S_1 does not match J . So, S_1 is not suitable.
- (ii) If only one of the applications that run on submeshes S_2 , S_3 and S_4 is swapped out, the available computing resources are not enough for J . Any two of submeshes S_2 , S_3 and S_4 can provide enough number of computing resources but the provided regions are not contiguous, which introduce further complexity if applied.

- (iii) When the application on submesh S_5 is swapped out, the obtained resources can provide a 3×3 submesh to J . But, only 9 out of 36 tiles are used by J . The other 27 tiles are not used and wasted. Moreover, the amount of memory needed to store the temporary values of the 36 suspended tiles is large. Thus, running J by preempting the application run on S_5 is very costly.

Support for the preemptive multi-tasking is costly, difficult and complex, if not impossible, on embedded manycore NoCs. Tessellation manycore OS [LKB⁺09] takes a preemptive multi-tasking approach. However, it is still at its early research stage.

6.2.3 Cooperative Multi-tasking for Systems with Multiple CPUs

As a parallel application has multiple CPU cores for running its tasks on embedded manycore NoCs, we propose to introduce cooperative multi-tasking for systems with multiple CPU cores where a running application manages its computing resources in terms of CPU cores rather than time slices. Further, a certain application can be designed with interruption points where the needed storage of temporary values is little. When an interruption point is met during its execution, the application may give up or take in CPU cores after negotiating with the OS. After that, the application can continue with a changed amount of resources and adjusted performance. Before meeting next interruption point, tasks of the applications run on CPU cores in a non-preemptive style.

An application that supports cooperative multi-tasking is called *cooperative*. A cooperative application can be designed to have a low bound (LB) for the number of CPU cores it uses such that the application can maintain the minimal performance for correct execution. On the other hand, a high bound (HB) indicates the desired resources for maximal performance. A cooperative application can run with n CPU cores ($LB \leq n \leq HB$) with variable performance.

Different from cooperative multi-tasking on a single-CPU system, it is not mandatory for all applications on embedded manycore NoCs to be cooperative. As embedded manycore NoCs are parallel systems with more than one CPU, the aforementioned cooperative multi-tasking can be realized if at least one application is willing to give up some of its CPU cores. Other applications that are not cooperative support the non-preemptive multi-tasking.

6.3 Hybrid Non-preemptive/Cooperative Multi-tasking

6.3.1 Overview

After the cooperative multi-tasking for systems with multiple CPU cores is introduced, we propose a novel hybrid multi-tasking approach which combines the desired features of non-preemptive multi-tasking and cooperative multi-tasking. It permits the co-existence of cooperative applications (applications that support cooperative multi-tasking) and non-cooperative applications (applications that don't support cooperative multi-tasking) running on the same embedded manycore NoC. Thereby, the proposed hybrid multi-tasking is named "hybrid non-preemptive/cooperative multi-tasking". It aims at enabling the interactions among applications and improving the flexibility of resource management at runtime. Though it is not mandatory, we also propose to integrate the hybrid multi-tasking into the submesh based resource management strategy presented in the preceding chapter.

The resource management under the proposed hybrid multi-tasking aims to implement the following features:

- When there are enough resources on embedded manycore NoCs for applications, the *GM* allocates resources to the applications such that they are executed with resources for maximal performance.
- When a new application comes requesting for resources and available resources are insufficient, the *GM* tries to obtain resources for the application via negotiations between the *GM* and running cooperative applications.
- When any application finishes and its resources are released, the *GM* communicates with running cooperative applications such that the ones, which are running with less resources and cannot achieve maximal performance, may gain more resources to achieve higher performance.

As each cooperative application can be a parallel program which runs on multiple compute cores, when it gives up control of its computing resources for an incoming application, it is designed to give up cores instead of time slices.

We elaborate below how the proposed hybrid multi-tasking helps overcome the failure of launching application shown in Figure 6.1. For the submesh S_5 , i.e., $PS_{3,3}^{6 \times 6}$, which is executing

the application App_5 , the tile $(3, 3)$ is chosen as the local manager. If the application App_5 is a cooperative application, its submesh $PS_{6,6}^{3 \times 3}$ can be cut for running the new application J and the resulted system configuration is shown in Figure 6.2.

The tiles left to the application App_5 can form two neighboring submeshes: $PS_{3,3}^{3 \times 6}$ and $PS_{6,3}^{3 \times 3}$. If the cooperative application App_5 is carefully designed and the local manager can limit the data communication of the application within the area formed by the two non-overlapped submeshes $PS_{3,3}^{3 \times 6}$ and $PS_{6,3}^{3 \times 3}$, which is left to the application after cutting a 3×3 submesh. There will be still no external communication contentions in the system.

Figure 6.2: An example of hybrid non-preemptive/cooperative multi-tasking

The key mechanisms in the proposed hybrid multi-tasking consist of the following:

- (i) A negotiation mechanism between GM and cooperative applications for obtaining or giving up resources. This mechanism is discussed in the following subsection.
- (ii) Architectural modifications to support bi-directional control messages. These modifications are discussed in Section 6.3.3 in detail.
- (iii) Methods for designing or parallelizing applications to support cooperative multi-tasking. In Section 6.3.4, we present such a method of designing cooperative applications or parallelizing applications into cooperative ones.

6.3.2 The Negotiation Mechanism

We discuss the typical interactions between the *GM* and an incoming non-cooperative application under the hybrid non-preemptive/cooperative multi-tasking as follows. Essentially, during these interactions, the *GM* and cooperative applications negotiate about resources.

- (i) When a new application *J* comes to the system, the global manager tries to find a suitable free submesh in the list *LIST*, which keeps the maximal free submeshes (discussed in the preceding chapter), to accommodate the application. If a free submesh of desired size can be identified, the global manager allocates it to *J* by executing the allocation algorithm.
- (ii) If there is no suitable free submesh for *J*, and if among the applications running in the system there is no cooperative application whose submesh can accommodate *J*, *J* has to wait. Otherwise, the system tries to negotiate for cooperative multi-tasking as below.
- (iii) The global manager broadcasts a “REQUEST” control message via the control network, which contains the information of the request by *J*, to the local managers whose submesh sizes are large enough for *J* to enquire if any cooperative application can give up a requested submesh for *J*. Before being processed by a local manager, a message from the global manager is received by network interface at the local manager and stored in a special buffer.
- (iv) The network interface of the tile that acts as the local manager of a non-cooperative application analyses the received message and sends a “NEGATIVE” control message back to the global manager to notify that the current application will not give up any resources. The network interface of a local manager can be configured to respond to incoming messages when an application enters the system.
- (v) A cooperative application checks resource requests at interruption points. After an interruption point is reached, a local manager analyses the received “REQUEST” message if there is one and sends a “POSITIVE” control message back to the global manager if it will give up the requested resources for the incoming application. Otherwise, the local manager sends a “NEGATIVE” control message to the global manager.

- (vi) If the messages from local managers received by the global manager are all “NEGATIVE”, the requested resources cannot be provided by cooperative application(s). Application J has to wait until some application is completed.
- (vii) If at least one “POSITIVE” message from a local manager is received by the global manager, there is some cooperative application that is willing to give up the requested resources. The global manager chooses one, if there are multiple “POSITIVE” messages, and sends a “CONFIRMED” message to the local manager whose “POSITIVE” message has been chosen.
- (viii) A “CANCELLED” control message is sent to each of the other cooperative applications that have not been chosen, if there are any, such that they can continue running after handling the “CANCELLED” message. If a previously received “REQUEST” message is still in the buffer when a “CANCELLED” message arrives, i.e., an interruption point has not been reached to consume the “REQUEST” message, the network interface removes the “REQUEST” message from the buffer.
- (ix) After the local manager of the chosen cooperative application receives the “CONFIRMED” message, it does the necessary preparation work, such as saving intermediate results, to concede a submesh of required size. Then, the local manager sends a “COMPLETE” message to the global manager. This message contains the information of the conceded submesh.
- (x) After the “COMPLETE” message is received by the global manager, the global manager starts J on the conceded submesh.
- (xi) When an application running on a submesh which was conceded from a cooperative application finishes, the submesh is released by the application and its control is passed to the global manager. When the global manager tries to return such a submesh to its original cooperative application if it still runs, the global manager sends a “REQUEST” control message to the cooperative application. The control message has a flag indicating returning resources.

The control messages used in the negotiation are listed in Table 6.1.

Table 6.1: Control messages used in negotiation

Message	Source	Destination	Meaning
REQUEST	global manager	local manager	Request for cooperation
NEGATIVE	local manager	global manager	Deny to cooperate
POSITIVE	local manager	global manager	Agree to cooperate
CONFIRMED	global manager	local manager	Cooperation is confirmed
CANCELLED	global manager	local manager	Request is cancelled
COMPLETE	local manager	global manager	Cooperation is completed

Note that all messages contain a message ID which can be used to match the responding messages to the request messages. The global manager guarantees the generation of correct message IDs.

If the incoming application running on a submesh conceded from a cooperative application finishes earlier than the cooperative application, the system will inform the cooperative application to take back the conceded submesh. Otherwise, if the incoming application running on a submesh conceded from a cooperative application finishes later than the cooperative application, the conceded submesh is taken back by the system using the deallocation algorithm and put into the free submesh list *LIST*.

6.3.3 Architectural Supports

As discussed in the preceding chapter, in the strategies [CM07] [CM08] [COM08] [CM10], the control messages are only sent from other tiles to the global manager and the control messages contain the status information of the tiles. The status information of a tile the address of this tile and an extra bit for showing whether the tile is busy or idle. Each control network in their implementation is not as a fully connected network but as a broadcast tree with reversed edges that accumulates the messages from the other tiles to the global manager, shown in Figure 5.3.

To support the hybrid non-preemptive/cooperative multi-tasking, our adopted platform has been extended from the platform in Figure 5.3 which was used in [CM07] [CM08] [COM08] [CM10]. Our adopted platform is shown in Figure 6.3. The major difference lies in the bidirectional control network and the network interface which are designed to enable interactions between the global manager and the local managers.

Figure 6.3: An embedded manycore NoC with a bidirectional control network

6.3.3.1 Modifications to Control Network

Besides receiving messages from local managers, the global manager also sends control messages to local managers, which is different from the unidirectional communication in [CM07] [CM08] [COM08] [CM10]. For this purpose, we add a physical channel to the control network which is represented by the dotted lines in blue.

Because the dimension of the NoC mesh can be large in embedded manycore NoCs and there is a need to transmit control messages, this information including the control messages and destination tile is sent in packets so as to save area and power consumption.

Considering the possibility of deadlocks, virtual channels have been designed in the control network. The number of virtual channels is configured to be at least 2. To avoid deadlocks, control messages from the global manager to local managers are always and only sent via the virtual channel with the lowest index, i.e., virtual channel 0. The messages from local managers to the global manager can only be sent through other virtual channels.

6.3.3.2 Modifications to Network Interface

The network interface for the control network is specially designed for the hybrid multi-tasking.

1) It has a buffer for storing incoming “REQUEST” messages before the cooperative application

reaches an interruption point to handle the “REQUEST” messages. 2) It also removes the corresponding “REQUEST” message in the buffer if a “CANCEL” message is received before the local manager reaches an interruption point to handle the “REQUEST” message. This situation happens when another cooperative application already has responded to the broadcast “REQUEST” message. 3) When an application is non-cooperative, the network interface of the tile where its local manager is located can be configured to automatically respond to a “REQUEST” message with a “NEGATIVE” message. In this way, the execution of the non-cooperative application is not disturbed.

6.3.4 The Method for Designing Cooperative Applications

An application should be well designed to support cooperative multi-tasking so as to further realize the hybrid non-preemptive/cooperative multi-tasking. In this section, we first discuss the features of cooperative applications. Then, we present a method for designing or parallelizing cooperative applications.

6.3.4.1 Desired Features of Cooperative Applications

According to the description of the steps of negotiation in realizing hybrid multi-tasking, the cooperative application must have the following features: 1) the cooperative application is a parallel application, i.e., has multiple tasks, such that it can utilize several cores to collaborate on computation. 2) one or more interruption points must be set in order that the application can check if there is a resource request when it reaches any of these interruption points. 3) the application must be able to continue execution to achieve correct results with different performance when some compute cores originally allocated to it have been conceded to another incoming application or a conceded submesh is returned back to it.

It would be difficult for any application to satisfy all above conditions. However, for certain category of applications, the above features can be satisfied by appropriately designing or parallelizing the applications.

6.3.4.2 A Method for Parallelizing Cooperative Applications

In this section, we present a method for parallelizing applications for cooperative multi-tasking when these applications have explicit or implicit data parallelism. As many applications have data parallelism, the application scope of this method is potentially broad.

In order to exploit data parallelism, the proposed method divides a data unit into several smaller parts and these parts of data are computed by several compute cores concurrently. The parts of data should be equal or approximate such that the workloads of compute cores can be nearly equal to achieve high performance. In this way, the first of the above features is satisfied.

Commonly, the applications with data parallelism need to process large amount of data one unit after another. After parts of data divided from a unit have been processed, the partial results should be combined together to obtain the correct final result. Then, the application continues to handle the next unit if there is one. An interruption point can be placed in the source code on the path after the partial results are combined and before a new unit is processed. Thus, the second of the above features can be met.

At the interruption points, if resources available to the cooperative application have been changed, the volume of partial data for each core can be adjusted according to the available resources. The correct result can still be achieved as the whole data unit is processed. The workloads of compute cores are still kept as balanced as possible. The achieved performance is also changed accordingly. With this, all above features are satisfied.

To implement the division and distribution of data, the cores used to execute the cooperative application are differentiated into two categories: master and slave, following the well-known *master and slave pattern* for parallel programming. There is only one core acting as master and it also takes the role of the local manager of submesh allocated to the application. The master is responsible for dividing a unit of data into approximate parts and distributing them to other cores which act as slaves. Slaves compute their partial data respectively and send partial results to the master. The master collects partial results and combines them into a whole result. After obtaining the whole result for the unit of data and before proceeding to the next unit, the master reaches the interruption point and checks if there is any resource request. After handling the possible resource request, the master proceeds to the next data unit if there is one.

6.3.5 Discussions

In this subsection, we want to highlight several important issues as follows.

In the above description of interactions, the incoming application is assumed to be non-cooperative for the purpose of simplicity. Actually, the incoming applications can be cooperative. For those scenarios, a cooperative application provides the minimal and optimal requested

sizes to the global manager. Once the minimal requested size is satisfied, the incoming application can get started. During its execution, the application is also able to take in more resources until the optimal requested size is satisfied. These scenarios can be handled by extending the above described negotiation mechanism of hybrid multi-tasking.

However, no application concedes its whole submesh when a new application presents a request. A non-cooperative application never responds to a request for resources. A cooperative application doesn't concede all of its resources for new applications and must have at least LB CPU cores for its execution. The number of conceded CPU cores from an application depends on the available resources. On the other hand, the OS guarantees that an application doesn't have more than HB CPU cores.

When a cooperative application concedes a smaller submesh from its submesh, the performance of this application will be lowered due to less resources. Even though, we don't want to introduce external communication contentions when a smaller submesh is cut from the submesh running the cooperative application. Hence, the left-top tile of a submesh is chosen as its local manager and the conceded smaller submesh is cut at the corner opposite to the local manager, i.e., the right-bottom corner. In this way, after the cutting, the tiles left to the cooperative application can be seen as two submeshes which are bordered with each other and there is no external communication contention from the conceded smaller submesh. When the cooperative applications are carefully designed and the local manager can somehow limit the data communication of the application within the two submeshes left to the application.

6.4 Evaluation of The Proposed Hybrid Multi-tasking

To evaluate the hybrid multi-tasking, an MPEG-2 encoder program is parallelized into a cooperative application whose behaviour is studied under resource cooperation requests at runtime.

6.4.1 Cooperative Application Example: MPEG-2 Encoder

6.4.1.1 Overview of MPEG-2 Encoder

Multimedia applications are important for not only general-purpose desktop processors but also embedded devices, including embedded manycore NoCs. MPEG-2 encoding is one of the popular media processing algorithms for coding videos.

Though there are more advanced multimedia technologies, such as MPEG-4 and H.264, these technologies have a common heritage similar to MPEG-2. As a result, the overall coding approach is quite similar and all these algorithms are based on a common heritage of block DCT transform based, predictive and entropy coding. Hence, the results from our study of MPEG-2 encoder could be extended and applied to other multimedia technologies.

An important aspect of the versatility of MPEG-2 is its layered structure. The hierarchy of layers in an MPEG-2 bit-stream is arranged in the following order: *Sequence*, *Group of Pictures (GOP)*, *Picture*, *Slice*, *Macro-block* and *Block*, shown in Figure 6.4. The different parts of the stream (except macro-blocks and blocks) are marked with unique, byte aligned codes called start-codes. These start-codes are used both to identify certain parts of the stream and to allow random access into the video stream. The random access ability is vital to parallelization.

Figure 6.4: The hierarchy of layers in an MPEG-2 bit-stream

The highest level in the layering is the sequence level. A sequence is made up of groups of pictures (GOPs). Each GOP is a grouping of a number of subsequent pictures. The purpose in creating such an identifiable grouping is to provide a point of random access into the video stream for play control functions (fast forward, reverse, etc.). Within each GOP are a number of pictures. Pictures are further subdivided into slices. Each slice is an encoding unit and is independent of other slices in the same frame. Particularly, each slice is a sequence of macro-blocks in raster scan order. A macro-block is 16×16 pixel group containing the luminance and chrominance data for those pixels in the picture. A macro-block is divided into blocks (6 to 12, depending upon format). A block is an 8×8 pixel group that describes the luminance or chrominance for that group of pixels. Blocks are the basic unit of data at which the decoder processes the encoded video stream. The relation of MPEG-2 data units is shown in Figure 6.5. Macroblocks and blocks do not have start-codes associated with them; their boundaries are discovered implicitly while decoding.

Frames in an MPEG stream are encoded into one of three types. All picture types use spatial correlation, but not all use temporal correlation. The first picture type, the intra coded picture (I-Frame), uses only spatial correlation. Since their decoding is independent of other pictures,

Figure 6.5: The layered structure of MPEG-2 data

I-Frames provide access points into the coded stream where decoding can begin. However, using just spatial correlation, they achieve only moderate compression. The second type of picture, the predictive coded picture (P-Frame), is coded more efficiently by also using temporal redundancies from a past I or P-Frame. These P frames are then used for reference in further prediction. The final picture type, the bidirectionally-predictive coded picture (B-Frame), uses temporal redundancies from both past and future reference pictures, and consequently achieves the highest degree of compression. B-Frames are never used as references for prediction.

Figure 6.6: The typical dataflow and components of an MPEG-2 encoder

As shown in Figure 6.6, the MPEG-2 algorithm is made up of the following processing blocks: motion estimation (*ME*) and motion compensation (*MC*), forward (*FDCT*) and inverse (*IDCT*) discrete cosine transforms, vector quantization (*Quant*) and inverse vector quantization (*iQuant*) and variable length coder (*VLC*). These tasks belong to the coder loop.

6.4.1.2 Comparison of Parallelization Methods

Task (functional) decomposition and data-domain decomposition are two commonly used methods for parallelizing applications. We compare them for parallelization of MPEG-2 as follows. We first introduce how these methods can be applied to the MPEG-2 encoder. Then, the advantages and disadvantages of these methods are discussed in terms of scalability and load balance.

As discussed in the above subsection, the MPEG-2 algorithm consists of several processing blocks. Therefore, each frame should experience a number of processing steps: motion estimation, motion compensation, integral transformation, quantization and entropy coding. The reference frames also need inverse qualification, inverse integral transformation, and filtering. When the functional decomposition method is applied, these functions could be explored for parallelization. Table 6.2 shows some major functions used by the single-threaded MPEG-2 encoder in ALPbench benchmark [LSA⁺05] and their average execution time in clock cycles, which are obtained on a simulator with a single PowerPC 405 core.

Table 6.2: The average execution time of different functions (in clock cycles)

Function	Average Executed Cycles	Function	Average Executed Cycles
pttransform	847,963,435	fdct1	838,396,034
ptmotion_estimation	436,240,518	frame_ME	435,348,305
frame_estimate	419,800,859	fullsearch	414,612,664
dist1	289,067,608	ptputpict	315,749,245
put_AC	204,728,614	put_intrablk	170,938,841
put_nonintrablk	112,264,377	ptitransform	45,194,449
idct	27,943,322	calcSNR	158,329,779
ptquant	21,473,231	put_bits	136,276,673

As shown in Figure 6.5, the MPEG-2 encoder can access different data granularities such as groups of pictures (GOP), pictures (frames), slices, macroblocks and blocks. If the data-domain decomposition method is applied, all these units are possible places to parallelize the encoder.

We first compare the two methods in term of the scalability. When the data-domain decomposition method is used, in order to increase the number of cores used in parallel processing, the size of the processing unit of each core can be decreased. Because of the hierarchical structure in GOPs, frames, slice, macroblocks, and blocks, many choices are available for the size of processing unit, thereby achieving good scalability. When the functional decomposition is applied, each core has a different function. To increase the number of cores, a function need

to be partitioned into two or more processes, based on their average execution time shown in Table 6.2, which can be executed on cores independently, unless the function is unbreakable.

Second, we compare the two methods in term of load balancing. With the data-domain decomposition method, each core performs the same operation on different data blocks that have the same dimension. In theory, without cache misses or other non-deterministic factors, all cores should have the same processing time. On the other hand, it is difficult to achieve good load balancing among functions, when the functional decomposition method is used, because the chosen algorithm determines the execution time of each function. Furthermore, any attempt to functionally decompose the video encoder to achieve a good load balance depends on algorithms, too. As the standard keeps improving, the algorithms are sure to change over time to exploiting parallelism at multiple levels to achieve a good load balance.

Figure 6.7: Data parallelism at slice level

6.4.1.3 Parallelization with Slice-level Data Parallelism

As each slice is an encoding unit and is independent of other slices in the same frame, slice-level parallelism is adopted, i.e., the data parallelism is exploited at the data granularity of slice. As shown in Figure 6.7, a frame or picture can be split into slices of equal size and these slices are further distributed to be processed by several compute cores. The processed partial results from these cores are then merged into a whole result.

However, parallelization based on slices has both advantages and disadvantages. The advantage lies in the independence of slices in a frame. Since they are independent, you can simultaneously encode all slices in any order. On the other hand, the disadvantage is the resulting increase in the bit rate. When a frame is divided into slices but quality is held at the same level, the bit-rate increases because slices break the dependence between macroblocks. The compression efficiency decreases when a macroblock in one slice cannot exploit a macroblock in another slice for compression.

Figure 6.8: Two slices with extra data to avoid increasing bit-rate

To avoid increasing the bit-rate at the same video quality and improve compression efficiency, we send partial data of macroblocks in neighbouring slices for compression. We explain this using the example shown in Figure 6.8. In the example, two slices are sent to a core and will be processed by it. Some data from the macroblocks from the above and below slices are sent to the core as well in order to improve the compression efficiency.

The three features for implementing cooperative multi-tasking, which are discussed in Section 6.3.4, can be satisfied by the parallelization of MPEG-2 encoder by exploiting data parallelism described in this subsection. Multiple cores cooperate to carry out the computation required by MPEG-2 compression algorithm. One of these cores acts as the master and it splits each frame of MPEG-2 bit-stream into slices. These slices are then distributed to other cores to be processed. An interruption point can be set at the source code executed by the master which is after all slices of a frame have been precessed and before the next frame is handled.

6.4.1.4 Implementation of MPEG-2 Encoder for Cooperative Multi-tasking

We choose the MPEG-2 encoder provided in ALPbench benchmark [LSA⁺05] which is modified from MSSG MPEG-2 encoder as our starting point. The parallel MPEG-2 encoder is

Figure 6.9: Flowcharts for master (a), slave (b) and output (c) processes

implemented as one master process and several subordinate processes. The subordinate processes can be further categorized as slave, which is responsible for data processing, and output, which is responsible for writing data into file. Particularly, these processes are created as MPI processes using the MPI library presented in Chapter 3. A core can only execute one process and different cores can be configured to be different processes. The flowcharts of these processes are shown in Figure 6.9 and they are discussed as follows.

As shown in Figure 6.9 (a), master process controls the execution flow of the parallelized MPEG-2 encoder program. During its initialization, it reads parameters and data from related files. It sends commands to slave processes to control their behaviour and sends initialization data to establish necessary data structures. Further, it sends initialization data to the output process. After initialization, it sends the partial data of a frame (slices) to slave processes for

processing. After sending slices to slaves, the master waits for returned data that are sent back from slaves when the processing of slices is finished. These returned data from all slaves are combined together to form a reference frame for the next frame.

After a slave process starts, it waits for the command from master to enter different processing tasks, shown in Figure 6.9 (b). Currently, two commands have been implemented and it is easy to add more commands to the code of slave. The implemented commands are “data parallel” and “exit”. When a slave performs the “data parallel” command, it executes the code of the processing blocks shown in Figure 6.6. Three differences are as below: 1) these processing blocks work on partial data of a frame, i.e., slices, instead of a whole frame; 2) a slave process sends reference data back to the master process; 3) a slave process sends compressed data to the output process. After a slave completes the execution of a task, it waits for a new command under the control of a loop. When a slave performs the “exit” command, it ends its execution.

After the output process starts, it waits for data of the sequence header from master. After that, output process waits for compressed data from slaves. After compressed data are received by output, following the correct order, they are merged into the data of a frame and written into the designated file.

Figure 6.10: The data flows between processes

As shown in Figure 6.10, various data are transmitted between processes. Assume that there are n slaves for data processing. These data are wrapped into messages which are sent over the

Figure 6.11: The sequence of sending/receiving messages between processes

NoC by calling the sending and receiving MPI APIs presented in Chapter 3. To avoid possible deadlocks in communication, the sequence of sending or receiving messages between processes has been carefully designed, shown in Figure 6.11.

To reduce the number of messages, multiple variables have been combined into a structure and the data of a structure is transmitted within a single message. However, there are situations that seem to contradict with the above idea of reducing number of messages. We can see that three messages have been sent from master to a slave before sending original and reference data of current frame. The reason for it is that the dynamic memory management is utilized in the code of slave. For example, message *A* comes to a slave followed by message *B*. The size of the buffer for receiving data that are contained as payloads of message *B* is contained in message *A* and transmitted to the slave. In this way, the buffer can be dynamically allocated by the slave with the required size.

6.4.2 Experiments and Experimental Setup

6.4.2.1 Overview of Experiments

We carry out two groups of experiments for hybrid non-preemptive/cooperative multi-tasking based on the parallelized MPEG-2 encoder discussed above.

One desired feature of a cooperative application is that it is a parallel application. Therefore, in the first group of experiments, discussed in Section 6.4.3, we evaluate performance of the parallelized the MPEG-2 encoder and its scalability. In detail, we obtain and analyse performance, i.e., speedups compared to the baseline execution using a single slave, of this encoder under different systems configurations. In each configuration, the number of slaves used in data processing is different.

The other desired features of a cooperative application are that it has interrupt point(s) and it can negotiate with the global manager on resource collaborations. In the second group of experiments, discussed in Section 6.4.4, the encoder is assumed to run in an environment supporting the hybrid non-preemptive/cooperative multi-tasking. We study behaviours of the encoder under variable computing resource collaboration requests.

We describe the experimental setup before discussing the above two groups of experiments.

6.4.2.2 Simulation Host

The experiments are carried out on a multi-core computer under low load. This machine has two Intel Xeon Quad Core X5460 CPUs with a clock speed of 3.16GHz and providing a total of 8 cores for computation. All cores share 12M L2 cache. The size of its main memory is 42GB. This machine runs Red Hat Enterprise Linux 5.

6.4.2.3 The Simulator

We generate a cycle-accurate simulator for an embedded manycore NoC with an 8×8 mesh whose architecture is described in Section 6.3.3. The microarchitecture of each tile is shown in Figure 6.12. This simulator is extended from the one described in Chapter 3. The MPI library and cross-compiler tool-chain presented in Chapter 3 are used here as well.

Similar to the data network, a pair of instructions are added to the PowerPC ISA in order to enable the PowerPC 405 core to access the network interface (NI_c) of the control network. These instructions are only executed when the interruption point is met. The PowerPC 405 core can read/send control messages from/to the network interface NI_c .

Figure 6.12: The microarchitecture of a tile

6.4.2.4 The Adopted Video Sequence

We use the Foreman YUV video sequence which is one of the commonly used video test sequences in the 4:2:0 YUV format which can be found at [ASU10].

The Foreman video sequence has 300 frames and we only use the first 30 frames. The types of these frames are as follows: *IPBBPBBPBBPBBIBBPBBPBBPBBPBBBB*. We use the version in CIF Format. Therefore, the size of each frame is 352×288 pixels. As the height of a slice is chosen to be 16 pixels, according the height of the frames, each frame has 18 ($288/16$) slices.

In each of the experiments, the parallelized MPEG-2 encoder processes the 30 frames and outputs a compressed file. The total size of raw YUV files of these frames is 4.35M bytes. The size of the compressed file is 103K bytes.

6.13.a: A snapshot of the original video

6.13.b: A snapshot of the compressed video

Figure 6.13: Screen snapshots of the original and the compressed videos

Figure 6.13 shows two screen snapshots of the original and the compressed videos respectively. We can discern that the quality of the picture of the compressed video is lowered due to compression.

6.4.3 Evaluation of The Parallelized MPEG-2 Encoder

In this group of experiments, we use eight different system configurations to execute the parallelized MPEG-2 encoder. The information of these configurations is shown in Table 6.3. These configurations use different numbers of slaves to process MPEG-2 stream data. The slices of a frame are dispatched to the slaves in order that they have approximate workloads. For configurations 4 and 5, the last slave has less slices to process than other slaves in the same configuration.

Table 6.3: Different configurations and achieved speedups

Configuration	No. of Slaves	Slices per slave	Total Execution Cycles	Speedup
1	1	18	115,394,400,037	1
2	2	9	57,982,530,640	1.99
3	3	6	39,081,195,803	2.95
4	4	5 ^[1]	32,692,186,617	3.53
5	5	4 ^[2]	26,287,315,780	4.39
6	6	3	19,876,577,323	5.81
7	9	2	13,539,963,701	8.52
8	18	1	7,331,024,925	15.74

[1]: Under this configuration, the last slave only processes 3 slices.

[2]: Under this configuration, the last slave only processes 2 slices.

The parallelized MPEG-2 encoder handles 30 frames under different configurations and the total numbers of cycles in executions are reported in Table 6.3. These results of execution cycles are visualized in Figure 6.14. It can be clearly seen from the figure that the total number of execution cycles for processing the frames decreases as the number of slaves increases.

We take configuration 1, where only 1 slave is used to process the MPEG-2 data stream, as the baseline to which other configurations are compared. We use symbol TEC_x to represent the Total Execution Cycles of a configuration x . The speedup of configuration x is defined as

$$Speedup = \frac{TEC_1}{TEC_x}$$

The speedups of different configurations are calculated and listed in Table 6.3. These results are visualized in Figure 6.15.

Figure 6.14: Total number of execution cycles under different configurations

Figure 6.15: Achieved speedups under different configurations

From Figures 6.14 and 6.15, we can see that the parallelized MPEG-2 encoder scales well following the number of slaves used in processing. However, due to synchronization and communication between the master and slaves, the achieved speedups are lower than the ideal speedups.

6.4.4 Experiments for Hybrid Non-preemptive/Cooperative Multi-tasking

At the beginning of the experiments, the parallelized MPEG-2 encoder is assumed to run on 20 cores which consist of 1 master, 1 output and 18 slaves. It is also assumed that among all applications running on the embedded manycore NoC only the parallelized MPEG-2 encoder supports cooperative multi-tasking.

During the compression of 30 frames by the encoder, some other applications come to the system. Under the request of cooperative multi-tasking from *GM*, the encoder gives up some

resources to let the incoming applications get started. The 30 frames are referred to as frames i , where $1 \leq i \leq 30$.

Table 6.4: Events in experiments

Experiment	Incoming application	Start Time*	Resource Request	End Time*
1	J_1	frame 4	3 cores	frame 16
1	J_2	frame 4	9 cores	frame 24
2	J_1	frame 4	3 cores	frame 24
2	J_2	frame 4	9 cores	> frame 30 **

*: The time is referred as the period during which a frame is under processing.

**: J_1 is still running when the processing of frame 30 is completed.

Some events occur during the processing of the MPEG-2 bit-stream. As shown in Table 6.4, during the processing of frame 4 by the parallelized MPEG-2 encoder, two applications J_1 and J_2 come to the system. They ask for 3 and 9 compute cores respectively.

Based on the data in above table, there are two experiments as follows. In experiment 1, J_1 finishes its execution during the processing of frame 16 and J_2 finishes its execution during the processing of frame 24. In experiment 2, J_1 finishes its execution during the processing of frame 24 and J_2 is still running when the processing of frame 30 is completed.

In these experiments, we also take the baseline system configuration where only 1 slave is used to process the MPEG-2 bit-stream. As the used simulator is cycle-accurate, we record the clock cycles spent on executing each of the 30 frames. The cycles for these frames are listed in the second column of Table 6.5. We also record the cycles used in processing each frame when the MPEG-2 encoder runs under hybrid non-preemptive/cooperative multi-tasking. In this way, the speedups for individual frames can be calculated.

6.4.4.1 Experiment 1 for Hybrid Multi-tasking

In Experiment 1, the execution of MPEG-2 encoder has several stages as follows. 1) The MPEG-2 encoder starts with 18 slaves and continues processing the frames from 1 to 4. 2) After the processing of frame 4 is completed, at the interruption point, the master checks and finds two “REQUEST” control messages for conceding cores from the global manager. After steps described in Section 6.3, the MPEG-2 encoder concedes a total of 12 (3+9) cores for incoming applications. 3) For frames from 5 to 16, the MPEG-2 encoder has 6 slaves to process the bit-stream. 4) After the processing of frame 16 is completed, the master checks and finds

Table 6.5: Statistics of processing individual frames in Experiment 1

Frame	Cycles with 1 slave (baseline)	Cycles with varying resources	Speedup for the frame
1	2,670,721,281	184,102,665	14.51
2	3,734,060,628	237,402,494	15.73
3	4,081,010,327	256,237,247	15.93
4	4,073,894,269	257,547,773	15.82
5	3,725,855,351	644,878,946	5.78
6	4,076,651,706	701,244,610	5.81
7	4,047,860,169	697,584,780	5.80
8	3,749,735,125	645,809,854	5.81
9	4,083,358,443	699,210,966	5.84
10	4,062,662,120	697,400,910	5.83
11	3,753,028,358	643,947,560	5.83
12	4,103,857,906	699,271,175	5.87
13	4,079,209,161	701,275,489	5.82
14	2,682,802,340	478,267,510	5.61
15	4,124,471,391	707,171,040	5.83
16	4,086,475,791	704,493,192	5.80
17	3,746,938,013	439,433,201	8.53
18	4,093,725,600	477,227,658	8.58
19	4,068,143,868	476,403,170	8.54
20	3,733,137,270	438,178,647	8.52
21	4,084,807,416	476,683,132	8.57
22	4,066,289,198	474,135,272	8.58
23	3,718,062,633	436,875,280	8.51
24	4,088,846,555	475,021,926	8.61
25	4,064,593,105	255,885,047	15.88
26	3,734,043,563	242,619,113	15.39
27	4,060,133,378	255,855,964	15.87
28	4,054,534,275	254,971,803	15.90
29	2,695,907,403	183,461,783	14.69
30	4,049,583,394	254,944,486	15.88
Total	115,394,400,037	14,387,738,219	8.02

one “REQUEST” control message for returning cores from the global manager. After being given back 3 cores, the encoder runs with 9 cores to process frames from 17 to 24. 5) After the processing of frame 24 is completed, the master checks and finds another “REQUEST” control message for returning cores from the global manager. After being given back 9 cores, the encoder runs with 18 cores to process frames from 25 to 30. The cycles for processing frames under varying resources are presented in the third column of Table 6.5. The fourth column of the table gives out the calculated speedups for individual frames. At the bottom of Table 6.5, we can see the total number of cycles used by the MPEG-2 encoder in processing the 30 frames.

The overall speedup in the experiment is 8.02 based on calculation.

Figure 6.16: Speedups for the individual frames in Experiment 1

The speedups of individual frames in Experiment 1 are also drawn in Figure 6.16. From this figure, we can clearly see that the speedup changes following the number of available compute cores, i.e., slaves.

6.4.4.2 Experiment 2 for Hybrid Multi-tasking

Figure 6.17: Speedups for the individual frames in Experiment 2

In Experiment 2, the execution of the MPEG-2 encoder has several stages as follows. 1) The MPEG-2 encoder starts with 18 slaves and continues processing the frames from 1 to 4. 2) After the processing of frame 4 is completed, at the interruption point, master checks and finds two “REQUEST” control messages for conceding cores from the global manager. After negotiation steps, the MPEG-2 encoder concedes a total of 12 cores for incoming applications. 3) For frames from 5 to 24, the MPEG-2 encoder has 6 slaves to process the bit-stream. 4) After the processing of frame 24 is completed, the master checks and finds one “REQUEST”

control message for returning cores from the global manager. After being given back 3 cores, the encoder runs with 9 cores to process frames from 24 to 30. 5) J_2 is still running when the processing of all 30 frames have been completed.

The speedups of individual frames in Experiment 2 are also drawn in Figure 6.17. The total number of cycles used by the MPEG-2 encoder in processing the 30 frames is 17,326,385,874. The overall speedup in the experiment is calculated to be 6.66.

6.4.5 Summary of Experiments for Hybrid Multi-tasking

Based on the first group of experiments, it is shown that the method for designing cooperative applications has been successfully applied to the MPEG-2 encoder, which has been converted into a parallel application that leverages the slice-level data parallelism.

Based on the second group of experiments, as shown in the Figures 6.16 and 6.17 for Experiments 1 and 2 respectively, the parallelized MPEG-2 encoder supports for cooperative multi-tasking and it dynamically adjusts its resources according to the resource allocation collaboration requests.

6.5 Conclusion

In this chapter, we have presented a technique for hybrid non-preemptive/cooperative multi-tasking to overcome the limitations of non-preemptive multi-tasking that has been adopted by existing works. We have introduced cooperative multi-tasking on embedded manycore NoCs and proposed an implementation of hybrid non-preemptive/cooperative multi-tasking. To support this hybrid multi-tasking, the architectural modifications and a method for parallelizing applications for cooperative multi-tasking are presented.

We have taken the MPEG-2 encoder as a case study and parallelized it for cooperative multi-tasking. Experiments show that 1) the method for designing cooperative applications has been successfully applied to the MPEG-2 encoder, which has been converted into a parallel application that leverages the data parallelism; 2) the parallelized MPEG-2 encoder has good support for cooperative multi-tasking and it dynamically adjusts its resources according to the resource allocation collaboration requests.

6.5.1 Novelty of Our Research

Different from existing strategies [CM07] [CM08] [COM08] [CM10] which adopt the non-preemptive multi-tasking, our work on hybrid multi-tasking proposes a novel way in enabling the interactions among applications at runtime such that these applications can collaborate with OS on resource management, which leads to flexibility and better resource usage.

The work on hybrid multi-tasking in this chapter is also different from the hierarchical strategy for resource management proposed in Chapter 5, which is responsible for resource allocation and deallocation. The hybrid multi-tasking enables the interactions among applications when applications are running, i.e., between the allocation process and the deallocation process.

Chapter 7

Runtime Thermal Management for NoC Based Manycore Systems

7.1 Introduction and Motivation

In preceding chapters, we have proposed submesh based core allocation schemes to schedule jobs when they come into NoC based multi-/many-core systems. Those schemes provide coarser granularity for the management of compute cores and reduce the complexity compared to management of individual cores.

Another important issue in multi-/many-core systems is thermal problems resulting from high power consumption. These could be thermal emergencies, localized hotspots and temperature gradients as discussed in the literature review. Many solutions to these problems come from different perspectives, but thermal problems relating to multiple cores in multi-/many-core systems in multiprogrammed environment have not been adequately addressed.

Like single-core processors, the temperature of each core of a multi-/many-core chip fluctuates under the conjoined effects of three factors: heat dissipation induced by the cooling system, heat exchange with neighbor cores, and heat generated by power consumption for computation. It is observed that, in a multiprogrammed environment, cores of a multi-/many-core chip under combinations of workloads can gradually have temporal and spatial temperature variations. The main reasons for the variations are as follows. 1) Different cores could work under different thermal stresses due to the variable power consumption during computation of one application and the distinct thermal profiles of different applications. 2) Even under same thermal stress, cores' temperatures may not arise to the same level if some are cooler [Ska03]. 3) Under temperature-unaware scheduling policies, one hot core may be scheduled to run applications before it cools down while other cores are idle and cool.

As the distinctive thermal profiles of different applications lead to temperature diversities on individual cores, i.e., some cores are very hot while others are cool, inappropriate core scheduling schemes without temperature awareness may worsen this situation by keeping some cores overheated while others are idle at the same time. Therefore, jobs should be scheduled to the cooler cores in order that hotter cores can have the opportunity to cool down. By doing this, heat can be balanced chip-wide eventually. This is the rationale behind temperature-aware schemes.

In this chapter, we propose temperature-aware submesh allocation schemes to schedule jobs when they come into multi-/many-core systems. The proposed schemes try to accommodate incoming applications with submeshes having favourable thermal features. These schemes use a strategy of preventing overheating and attempt to balance heat throughout the whole chip. Eventually, our schemes help to balance heat throughout the multi-/many-core chips so that we expect the multi-/many-core systems have longer device lifetime and higher reliability [Lu04].

A scheme for temperature-aware contiguous submesh allocation is proposed to help achieve heat balance over the whole chip in a multiprogrammed environment. The proposed scheme is a contiguous scheme for temperature-aware submesh allocation and is a natural extension to existing contiguous submesh allocation schemes that considers thermal factors. The essence of this scheme is scheduling jobs to contiguous submeshes at cooler locations in a multi-/many-core system according to temperature-aware strategies. This scheduling is an aforethought and preventive approach and reduces the chances of thermal problems' happening.

Some cores of multi-/many-core processors might eventually become overheated, i.e., their temperatures are quite high such that their soft error rate is too high for them to perform correct calculations. These overheated cores have to be stopped from computation temporally in order to cool down. During the suspended periods, the overheated cores can be regarded as "temporally faulty". When these cores cool down, they can be used for computation again.

The scheme for temperature-aware contiguous submesh allocation presented above does not work when there are "temporally faulty" cores as the scheme assumes that all cores are available and allocates contiguous physical submeshes. New techniques need to be developed for submesh allocation when some cores or tiles of mesh are faulty. Therefore, a new form of "virtual submeshes" is introduced to handle allocations of resources with existence of overheated cores. Algorithms using the dynamic programming technique are proposed to construct virtual submeshes which satisfy the computational requirements and are thermally favourable.

Due to the possible existence of overheated cores when applications come to the system, an adaptive scheme is presented for submesh allocation that identifies contiguous physical submeshes or virtual submeshes for incoming applications. The set of resources allocated to an application by the adaptive scheme is called a “partition”. When there are no overheated cores on the system, the scheme performs the same as for contiguous submesh allocation. When overheated cores appear, the adaptive scheme constructs virtual submeshes that satisfy the computational requirements of the applications and allocate physically contiguous submeshes that contain the constructed virtual submeshes to the applications. Therefore, when overheated cores appear, there are extra cores within the partition of an application.

Extra cores with a partition don’t necessarily mean wastage of resources. On the one hand, according to “utilization wall”, in the deep sub-micron region, the number of CPU cores on a manycore processor can be large and not all of them can be powered to work simultaneously. Shutting down some overheated cores enables us to power on some other cool cores. Meanwhile, the suspended overheated cores could cool down. On the other hand, the redundant non-overheated cores can be used to further achieve heat balance by migrating computation from hotter cores to cooler cores. In this way, redundant non-overheated cores inside a partition become benefits instead of being considered merely as “wasted”.

Therefore, a runtime temperature-aware task migration technique, is further suggested to reduce temperature variations among cores inside the partition of an application during its execution if possible, by exploiting the redundant non-overheated cores. This “intra-partition” task migration will not interfere with other applications running on the same chip because applications run on physical submeshes that do not overlap.

The adaptive scheme for submesh allocation and the “intra-partition” temperature-aware task migration technique together provide a relatively complete runtime thermal management solution for multi-/many-core processors which protects chips from potential damage due to thermal crises during execution of applications in a multiprogrammed environment. However, when the combination of above techniques cannot effectively protect chips, other thermal management techniques such as “stop-go” have to be used.

Our proposed runtime thermal management techniques can also serve as complements to other runtime thermal management techniques. For example, after one application has been treated for power density reduction using methods provided in [Nar05][Nar06], our schemes can further balance the temperatures at runtime by allocating appropriate cores for it.

7.1.1 Contributions and Chapter Organization

Contributions of this chapter are as follows:

- a scheme for temperature-aware contiguous submesh allocation is proposed to include thermally favourable cores in forms of submeshes when resources are allocated;
- a new form of virtual submesh, i.e., “temperature-aware virtual submeshes”, is introduced for the appearance of overheated or faulty cores;
- an adaptive scheme for temperature-aware submesh allocation and a solution adopting combined proactive and reactive strategies are proposed for runtime thermal management.

The rest of this chapter is organized as follows. Section 7.3 presents a scheme for temperature-aware contiguous submesh allocation. The concept of temperature-aware virtual submesh is introduced in Section 7.4 and two algorithms for constructing thermal favourable virtual submeshes are also presented there. An adaptive scheme for submesh allocation is shown in Section 7.5. A temperature-aware task migration technique which complements the adaptive scheme is presented in Section 7.6.

7.2 Preliminaries

7.2.1 Several Terminologies about Submesh

Submesh-based processor allocation schemes have been successfully applied to large parallel and distributed systems with mesh topology. Many contiguous and non-contiguous schemes have been discussed in the literature review. Here, as we developed our proposed scheme for contiguous submesh allocation based on a good existing scheme [Aba06], a brief introduction to this scheme and several terminologies used in it are necessary for our further discussion.

Switching request orientation is supported if a request for the allocation of an $a \times b$ submesh can be satisfied by a $r \times s$ free submesh, such that $(r \geq a \text{ and } s \geq b)$ or $(s \geq a \text{ and } r \geq b)$.

A *target submesh* is the physical submesh allocated to an incoming application. An *allocation submesh* is a suitably-sized free physical submesh from the list on which a target submesh is expected to be allocated. Several physical submeshes at the corners of an allocation submesh which satisfy the request of the application at the same time are called *candidate submeshes*.

If request orientation switch is supported in allocation scheme, the number of candidate submeshes in an allocation submesh can be up to 8.

A free submesh with the largest *boundary value* is chosen from *candidate submeshes*. The boundary value of a free submesh is defined as the sum of the boundary values of the tiles located on its periphery, where the boundary value of a tile is the sum of the number of allocated neighbor tiles and the number of mesh edges on which the tile lies. Allocating free submeshes with largest boundary values leads to *compaction*, i.e., higher resource efficiency.

The free-list submesh allocation scheme in [Aba06] maintains an unordered list of possibly overlapped free submeshes. When a request comes, the first size-suitable free submesh is selected as the *allocation submesh* for the request. Each allocation submesh has up to eight *candidate submeshes* at its corners with different orientations. Then one with the largest boundary value is chosen from *candidate submeshes*. After allocation and deallocation, the scheme maintains the free list to keep the maximal submeshes in it.

7.2.2 Acquisition of Runtime On-chip Temperature

Knowledge of runtime on-chip temperature is critical for runtime thermal management techniques. Several methods have been proposed to acquire the on-chip temperatures.

On-chip thermal sensors were adopted in modern processors in order to get temperature measurements. For example, there are two independent thermal sensors in the Intel Pentium 4 Processor [Car01]. IBM POWER 5 processor employs 24 on-chip thermal sensors to track the run-time temperature of each power-hungry component [KST04a].

A technique is proposed in [CS06a] to get accurate realtime temperature by using a linear formula, which is acquired by an off-line regression analysis, about activity data collected from performance counters and temperature. In most cases, only the activity count of the functional unit that is investigated needs to be considered. Using this formula, temperatures of functional units can be calculated with very small overheads.

A method to measure power, thermal and performance of real microprocessors was reported in [MMBNBR07] where thermal maps are captured using infra-red cameras with high spatial resolution and high frame rate.

A technique called software thermal sensor (STS) [WJY⁺07] was proposed to track chip temperature through an OS resident software module that generates live power and thermal

profiles of the processor using highly efficient numerical methods to minimize the overhead of temperature calculation and an efficient algorithm for functional unit power modeling.

A framework consisting of several techniques is proposed in [LMMM08] to create sensor infrastructures for monitoring the maximum temperature on a multicore system. An interpolation scheme is used to estimate the maximum core temperature through interpolation of the sensor readings collected at the static grid points. Further, a dynamic scheme where only a subset of the sensor readings is collected is used to predict the maximum temperature of cores.

The problem of estimating the actual temperature of on-chip thermal sensor when the sensor reading has been corrupted by noise is addressed and a statistical methodology is proposed in [ZS09] to predict the actual temperature for a given sensor reading.

In simulator based studies, the runtime on-chip temperatures of functional units on processors can be calculated by using the *HotSpot* simulation module proposed in [Ska03] [SSS⁺04] with runtime power traces within micro-architectural functional units and the physical implementation parameters of these micro-architectural units. The *HotSpot* temperature model is an accurate yet fast model based on an equivalent circuit of thermal resistances and capacitances that correspond to micro-architecture functional units and essential aspects of the thermal package. The model is validated using finite element simulation.

During simulations, the runtime power traces of functional units on a chip can be obtained by using the power simulation modules such as *Wattch* [BTM00] or *McPAT* [LAS⁺09]. In order that power/performance tradeoffs can be made visible to chip architects, *Wattch* is proposed for analyzing and optimizing microprocessor power dissipation at the architectural level. The power estimations by *Wattch* are based on a suite of parameterizable power models for different hardware structures and on per-cycle resource usage counts generated through cycle-level simulation. Recently, *McPAT*, which is a new modeling framework, has been developed to support design space exploration for processor configurations ranging from 90nm to 22nm and beyond. *McPAT* also provides power results when it works with performance simulators.

The process of calculating temperature using *HotSpot* consists of the following steps. 1) When a simulation starts, the information of floorplan and the initial temperatures of micro-architectural units are provided to the simulator. This floorplan information consists of an adjacency matrix describing the size and position of units on the floorplan. 2) Power trace is captured when programs are executed. One power/performance simulator, which can be obtained by integrating the power model such as *Wattch* or *McPAC* into a cycle-level performance

simulator, can be used to provide online power trace of program. Note that HotSpot is completely independent of the choice of power/performance simulator. 3) With the data of power trace, floorplan and current temperature of micro-architectural units, the Hotspot simulation module receives the averaged power dissipated in each unit, which is generated by the power module, over a user-specified interval and calculates the newly computed temperatures using the RC solver at runtime.

7.3 A Scheme for Temperature-aware Contiguous Submesh Allocation

7.3.1 Temperature-aware Allocation and Deallocation Algorithms

As discussed before, there are several reasons to apply submesh allocation schemes into the future multi-/many-core systems and we repeat them here for better readability. Firstly, future multi-/many-core systems could possibly be 2D-mesh connected since they comprise many homogeneous cores having almost same layouts. The regularity of mesh and submeshes can make management of cores easier. Secondly, submesh is a reasonable granularity for cores allocation. A job usually consists of many tasks and these tasks can be executed on several cores in parallel to get maximal performance. Thirdly, submesh is a contiguous area where cores are in close propinquity which keeps communication within a submesh cost low and internal.

Besides justifications for our proposed scheme, the low-cost and accurate realtime temperature measuring method [CS06a] makes it feasible. We only need to measure temperatures of the hottest functional units in cores like integer registers [CS06b]. The temperatures are of help to decisions of scheduling policies.

We adopt algorithms from [Aba06] and make two improvements for temperature awareness. First, temperature-aware heuristics or policies are used to single out one proper submesh. Second, a deallocated submesh is always appended to the end of the list of free submeshes. The second extension tries to give the newly released cores more time to cool down since local policies pick an allocation submesh starting from the head of the list.

When one job comes into a multi-/many-core system, the allocation algorithm is invoked to find one candidate submesh of cores from submeshes in the free list; after one job finishes, the deallocation algorithm is called to release the cores and allocated submesh. Both algorithms rearrange free submeshes for maximum-sized submeshes.

The premise for one successful allocation is that there are enough unoccupied cores and at least one size-suitable submesh for the job. So, in our allocation algorithm, the first step is checking resource availability. Then it calculates the temperatures of cores and updates related variables. The outline of the allocation algorithm and the deallocation algorithm are given in flow charts shown in Figure 7.1 and Figure 7.2 respectively.

Our allocation algorithm with local policies is the same with the allocation algorithm in [Aba06] except for the candidate submesh choosing criteria. Our allocation algorithm with global policies is the worst case for its counterpart. Our deallocation algorithm differentiates its counterpart with the position of released submesh in list. The different steps only take $O(1)$ time. Therefore, the time complexity of our algorithms is $O(f)$ time, where f is the number of submeshes in the list; the space complexity of our algorithms is $O(f)$.

7.3.2 Temperature-aware Allocation Policies

There are two options to choose one *candidate submesh* from a particular *allocation submesh*. One is scheduling jobs to the coolest candidate submesh in the allocation submesh. The rationale is that most power consumption is needed to get its cores' temperatures increased to a certain value among all candidate submeshes. The other is scheduling jobs to the candidate submesh whose neighbor cores are at lowest temperature. The reason is that the coolest neighbor cores may perhaps take most heat away.

There are also two choices to select an *allocation submesh* from the free list of submeshes according to their locations. One is choosing the first size-suitable free submesh from the head. The reasons are that it has the shorter time than searching the whole free list and the selected submesh possibly has been cooling down for longest time. The other is searching the whole free list and trying to find out the globally optimized one. These two choices lead to different algorithmic complexity.

It is highlighted that management of free list for maximum sized submeshes performs submesh subtractions and expansions. These actions may cause the cores in the same submesh having different idle time during which the cores can cool down.

Accordingly, we investigate the following policies:

- (i) **Resource Optimization policy (R.O.):** This policy aims at compaction of allocated submeshes with cores. The first size-suitable submesh is selected as the allocation submesh.

Figure 7.1: The allocation algorithm

Among its candidate submeshes, the first one with the largest boundary value is picked out for the request. This strategy is not temperature-aware and it possibly allocates the cores which are quite hot and need to be cooled down.

- (ii) **First Coolest policy (F.C.):** The first size-suitable submesh is selected as the allocation

Figure 7.2: The deallocation algorithm

submesh. The algorithm gets the highest current temperature of the cores in every candidate submesh and takes this value as its *candidate submesh temperature*. Among all the candidate submesh temperatures, the lowest one is selected as *allocation submesh temperature*. The corresponding candidate submesh is selected for the request.

- (iii) **First Neighbor-aware policy (F.N.):** The first size-suitable submesh is selected as the allocation submesh. The highest current temperature of neighbor cores to the border of one candidate submesh is taken as its *neighbor temperature of candidate submesh*. The lowest *neighbor temperature of candidate submesh* of all candidate submeshes is chosen as *neighbor temperature of allocation submesh*. This candidate submesh is allocated to the request.
- (iv) **Global Coolest policy (G.C.):** All size-suitable submeshes are allocation submeshes. Our algorithm evaluates all *allocation submesh temperature* and selects the lowest one as the *global coolest allocation submesh temperature*. The candidate submesh whose *candidate*

submesh temperature is the *global coolest allocation submesh temperature* is chosen for the request.

- (v) **Global Neighbor-aware policy (G.N.):** All size-suitable submeshes are allocation submeshes. All *neighbor temperature of allocation submesh* are evaluated and the lowest one is chosen as the *global neighbor temperature of allocation submesh*. The candidate submesh whose *neighbor temperature of candidate submesh* is the *global neighbor temperature of allocation submesh* is chosen for the request.

Figure 7.3 shows a runtime snapshot of a 64-core manycore chip in the form of an 8×8 mesh assuming that a tile has only one core within itself. Symbol $PS_{i,j}^{r \times s}$ is used to represent one submesh with the integers i and j indicating the X, Y position of its left-top corner and the integers r and s indicating the submesh's width and height in terms of tiles. For example, submesh S_1 which can be annotated as $PS_{1,1}^{4 \times 2}$ and submesh S_4 which can be annotated as $PS_{5,2}^{1 \times 4}$ are occupied by two jobs respectively.

Figure 7.3: Example for allocation policies

Five jobs are running on five submeshes from S_1 to S_5 which are shadowed areas with circles in dotted lines in Figure 7.3. After above five submeshes are allocated, there are six maximum-sized submeshes in the free list: $PS_{3,1}^{8 \times 2}$, $PS_{3,3}^{6 \times 3}$, $PS_{3,1}^{1 \times 4}$, $PS_{3,3}^{2 \times 6}$, $PS_{1,5}^{3 \times 5}$, $PS_{2,5}^{4 \times 4}$.

Table 7.1: Policies and their scheduling results

Policy	R.O.	F.C.	F.N.	G.C.	G.N.
Allocation Submesh	$PS_{3,1}^{8 \times 2}$	$PS_{3,1}^{8 \times 2}$	$PS_{3,1}^{8 \times 2}$	$PS_{2,5}^{4 \times 4}$	$PS_{3,1}^{8 \times 2}$
Candidate Submesh	$PS_{3,1}^{2 \times 2}$	$PS_{3,7}^{2 \times 2}$	$PS_{3,7}^{2 \times 2}$	$PS_{2,7}^{2 \times 2}$	$PS_{3,7}^{2 \times 2}$
Position	A	B	B	C	B
Values	5/51.1°C ¹	48.3°C ²	51.5°C ³	47.8°C ⁴	51.5°C ⁵

- 1: The largest boundary value is 5. temperature is 51.1°C
- 2: Allocation submesh temperature
- 3: Neighbor temperature of allocation submesh
- 4: Global coolest allocation submesh temperature
- 5: Global neighbor temperature of allocation submesh

Now, we assume that one job comes and requests a 2×2 submesh, which can be annotated as $R^{2 \times 2}$, for its execution. It is also assumed that the order of candidate submeshes at four corners in an allocation submesh is clockwise. Based on the above, the system makes the decisions and the different policies' allocation results for this request $R^{2 \times 2}$ are listed in the Table 7.1.

We can make several observations as follows based on Table 7.1.

- The R.O. policy does not consider thermal factors and often leads to bad solutions. It is a local policy since it picks out a candidate submesh from the first size-suitable allocation submesh.
- The policies like F.C. and F.N. also can only choose the local optimized candidates. Due to the submesh subtractions and expansions during management of free list, some cores in the first submesh may not be the ones which have been idle for the longest time. But these policies are faster.
- The global policies like G.C. and G.N. have the biggest possibility to find chip-wide temperature-optimized submeshes. But they need more time to accomplish.

7.3.3 Evaluation of The Proposed Scheme

A parallel multicore simulator [DM06a] extended from SimpleScalar PowerPC simulator (SSPPC) [SNKB01] is used as a basis simulator to evaluate the proposed scheme and its scheduling policies. This simulator integrates Wattch and HotSpot 2.0, which have been introduced in Section 2.5.3.

Figure 7.4: NoC architecture and cores

7.3.3.1 Simulator Setting

The tiles (cores) of the multi-/many-core chip are assumed to be connected via an NoC infrastructure. Here, each tile has one core inside it. We extend our layout for 16 cores in a way similar to the one used in [Nar06] and the cores are connected in a 4×4 mesh topology. The size of the mesh could be arbitrary, such as 8×8 , 16×16 , 32×32 , and etc. However, the bigger the size of the mesh used, the more resources and time are required to run the simulations. The size of 4×4 is enough to show our scheme and its policies. We use a floorplan of cores similar to that used in [DM06a] and reduce the core size accordingly. Each core has various components necessary for an out-of-order pipeline. There is one network interface unit and a router attached to each core. Figure 7.4 shows the architecture of the NoC and the floorplan of a core with a router. Here, a network interface, which connects the components inside the tile and the router, is omitted to keep the diagram simple.

Several modifications have been made to the original SSPPC simulator. The shared Level-2 cache (16M) is removed and individual Level-2 cache (1M) is added to each core; additional functionality is implemented to run programs on particular cores; the temperatures are recorded for continuous tasks in workloads. The cores' temperatures at the end of one run are saved into files; in the following run, the initial temperatures of cores are read from these files. For the first jobs in the workloads, the initial temperature of cores is set to 45°C .

7.3.3.2 Workloads

Continuous workloads similar to the workloads in [DM06b] comprising many jobs and their properties are generated randomly. Every job has the following properties: *size of request* (width and height of submesh of cores for the job), *arrival time* (the time the job arrives), *programs running on cores* (randomized combinations of benchmark programs) and *randomized cycles of programs to run*.

The width and height of the submesh are assumed to have uniform distributions over $[1, W]$ and $[1, H]$; W and H are width and height of NoC mesh respectively. Jobs are also assumed to have exponential inter-arrival times.

The programs used in the simulation are selected from SPEC CPU2000 benchmark suite. They are *bzip2*, *gap*, *gzip*, *applu*, *mesa*, *mgrid*, *swim* and *wupwise*. They run with the reference inputs of SPEC CPU2000. The cycles of programs to run are randomized to further represent variable thermal pressures.

7.3.3.3 Simulation Procedure

The SSPPC simulator is invoked a child process of the allocation algorithm program. The simulator subsequently generates several child processes that accept the requests from the parent process of running certain programs on specified cores with parameters indicating the workloads' features. The transient temperatures of individual functional units are recorded at intervals of 100000 cycles.

Due to the different thermal pressure on functional units, there are also temperature variations among them. We choose the highest temperature of units as the current temperature of the core. These values are fed back into our allocation scheme to make scheduling decisions.

7.3.3.4 Evaluation Metrics and Results

The highest temperature of a core or its peak temperature is one critical evaluation metric here. A higher temperature than the threshold temperature causes failure in a short time. The other two metrics are temporal variance and spatial variance.

Temporal variance and spatial variance are defined to evaluate the allocation policies. N cores $\{c_i | i = 1, 2, \dots, n\}$ in a multi-/many-core chip, at sampling count t , have temperatures $f(c_i, t)$ respectively. Suppose time t starts from 1 to T , i.e., there are T sampling times. The

average temperature of c_i during the period is $\overline{c}_i$. The average temperature of cores at the time of t is $\overline{s}_t$.

(i) *Temporal Variance(T.V.)* of a multi-/many-core chip is defined as follows:

$$T.V. = \frac{1}{n} \sum_{i=1}^n \sqrt{\frac{1}{T} \sum_{t=1}^T (f(c_i, t) - \overline{c}_i)^2}$$

(ii) *Spatial Variance(S.V.)* of a multi-/many-core chip is defined as follows:

$$S.V. = \frac{1}{T} \sum_{t=1}^T \sqrt{\frac{1}{n} \sum_{i=1}^n (f(c_i, t) - \overline{s}_t)^2}$$

The above metrics evaluate the temperature variances of cores on the scope of whole chip. Lower variances indicate that the temperatures of cores throughout the chip are more balanced.

7.3.3.5 Simulation Results

We report simulation results with a workload of 10 general tasks for five policies. Each task comprises one or more benchmark programs. The workload takes equal time to finish under these policies. The difference is that tasks in the workload are scheduled to run on different cores according to the policies and the transient temperatures of cores. The details of workload are shown in the the Table 7.2.

Table 7.2: A continuous workload and its tasks

task	submesh size	benchmarks	cycle numbers
task1	2×1	swim,wupwise	1.47×10^9
task2	2×2	gap,mgrid,gzip,swim	1.79×10^9
task3	2×1	mesa,applu	1.42×10^9
task4	2×1	gap,gap	1.69×10^9
task5	2×1	applu,gap	1.62×10^9
task6	2×1	swim,bzip2	1.32×10^9
task7	2×1	gzip,applu	1.61×10^9
task8	1×1	mesa	1.28×10^9
task9	3×1	wupwise,swim,bzip2	1.80×10^9
task10	2×1	wupwise,gzip	1.19×10^9

In Table 7.2, the order of benchmark programs in one task should be followed when the algorithms schedule the programs to cores. The intervals between the tasks in our simulations are set to zero for simulating the worst-case situations for the R.O. policy where the cores around the left-bottom corner are always chosen leaving insufficient time for them to cool down.

The peak temperatures of the cores under the aforementioned workloads are listed in the Table 7.3. We can see from the table that under the thermal pressures of tasks in a continuous workload temperature-aware policies can help cores avoid high temperatures by allocating cooler cores to incoming tasks.

Table 7.3: Peak temperatures under policies

policy	R.O.	F.C.	F.N.	G.C.	G.N.
Peak temperature	103.09°C	85.03°C	82.12°C	80.11°C	79.86°C

The results of temporal variance and spatial variance under five policies for the same workload are shown in Figure 7.5. Variances under five policies are normalized as percentages of the largest one. For example, in Figure 7.5, the temporal variance under the F.C. policy is the largest one and the one under F.N. is around 94.6% of it.

Figure 7.5: Normalized variances

Figure 7.5 shows that global temperature-aware allocation policies generally lead to smaller temporal variance and spatial variance compared with local policies. For temporal variance, the reduction is between 2.9% and 10.2%. For spatial variance, the reduction is from 3.9% to 14.1%. Among the local policies, R.O. is not always the worst since it can possibly allocate cores on the border of the chip where the cooling system takes lot of heat away. This can be seen from Figure 7.5. It is hard to tell the better one between coolest policy and neighbor-aware policy since both heat from power consumption and heat from neighbor cores could take the dominant role in one core's temperature rising. It depends on the thermal profiles of applications and the runtime temperature situation.

7.4 Temperature-aware Virtual Submeshes

7.4.1 Motivational Example

As discussed above, some cores could eventually become overheated, i.e., their temperatures are enough to cause soft errors. These overheated cores have to be suspended from computation to cool down and can be regarded as “temporally faulty”. Figure 7.6 shows a Network-on-Chip which has 25 tiles connected as a 2D 5×5 mesh. It is assumed that each tile consists of only one compute core and other uncore components. In the following discussion, core and tile are exchangeable.

In this chapter, thermal issues are of primary interest to us. For simplicity, details of tiles and physical links between them are abstracted away. Each tile is represented by a square. The highest transient temperature of functional units in a tile is defined as the *temperature of a tile*. Frequently, the units that have highest temperatures lie inside cores, for example, register files [Ska03]. The temperature of a tile u is denoted as $temp(u)$. The temperatures of tiles are labeled on squares as numbers respectively. In this example, a tile u is treated as overheated if $temp(u) \geq 100^\circ\text{C}$. The NoC in Figure 7.6 has six overheated cores, which are represented as solid squares in red.

Figure 7.6: A Network-on-Chip with six overheated cores

To handle processor allocation when there are faulty nodes in the mesh, a virtual submesh allocation was proposed in [KY98] to provide virtual submeshes to incoming jobs when some nodes of a mesh are faulty. In this scheme, the rows and columns of a mesh which contains any faulty node are removed from the mesh. The resulting mesh which consists of the remaining rows and columns are used for submesh allocation.

However, this scheme in [KY98] leads to huge wastes of computing resources. Let us explain this via an example, shown in Figure 7.7. In 5×5 mesh of Figure 7.7, the six overheated cores are represented as solid squares in red. If the scheme in [KY98] is applied, all rows and all columns are removed and no submesh can be provided. In this example, all 19 non-overheated or non-faulty cores are wasted.

Figure 7.7: A virtual submesh formed by deleting rows and columns

To overcome the above limit of the scheme in [KY98], we introduce a new form of virtual submesh that is discussed in the following subsection. Our aim is to better utilize the non-faulty cores to construct virtual submeshes of larger sizes compared to the scheme in [KY98].

7.4.2 A New Form of Virtual Submesh

We introduce a new form of virtual submeshes. In such a virtual submesh, cores belonging to a virtual column (row) may come from different physical columns (rows). Overheated cores whose temperatures are above a threshold value are treated as temporally faulty and are not considered for resource allocation.

In [KY98], when processor nodes are faulty, all resources inside them are assumed to be unavailable before recovery. The determination of virtual submesh in [KY98] is simplistic and it removes columns or rows of processor nodes from computation (these nodes are called *bridge nodes*) whenever there is a faulty node on them and *bridge nodes* are only used as bypass links. This could lead to waste of computing resources and is perhaps due to unavailability of routers in the faulty processor nodes.

The new form of virtual submesh proposed by us is quite different from those in [KY98] in following aspects. In our proposed virtual submesh, routers inside overheated tiles and the links connected to them are assumed to work normally and only compute cores are shutdown to cool. This assumption is reasonable due to the following facts: the overheated functional units are commonly inside compute cores, for example, register files [Ska03]; fine-grained power management techniques implemented in manycore chips, such as the Intel 80-tile chip [VHR⁺07], allow some functional units of a tile to be switched off instead the whole tile. Discussion on faulty routers is out of the scope of this dissertation. Therefore, availability of routers in overheated tiles enables us to adopt the different concept of virtual submesh.

We elaborate on the concept of virtual submesh with aid of the above example shown in Figure 7.6. If a job J with allocation request of a 5×3 submesh comes, if the contiguous submesh allocation scheme presented in the preceding chapter is applied, a suitable-sized physical submesh can not be found and J has to be kept in queue waiting for resources. The faulty mesh scheme in [KY98] cannot provide any submesh, shown in Figure 7.7, and also has to keep J waiting. For non-continuous schemes, J cannot be accommodated either.

Figure 7.8: Two virtual submeshes formed by virtual columns

We observe that the number of non-overheated cores is 19 which is enough for request J . As shown in both diagrams of Figure 7.8, 15 cores are picked out to form three columns (connected by dotted lines) and these cores can satisfy the computation request of job J . These columns are called “virtual columns” since the dotted lines indicate the logical connections between cores and accordingly these virtual columns form a “virtual submesh”. In such a virtual submesh, cores belonging to a virtual column (row) may come from different physical columns (rows).

The flexibility of a virtual submesh is at the cost of increased communication latency compared to a physical submesh. In a physical submesh, the number of hops which a flit traverses between neighboring cores is 1. While in a virtual submesh, the above number of hops could be bigger than 1, depending on the distance of logically connected cores. For example, in Figure 7.8, the number of hops which a flit traverses from $core_{i+1,j+1}$ to $core_{i+2,j+2}$ is 2 (Assume that the core at the left-top corner is annotated as $core_{1,1}$). Fortunately, techniques deployed on NoC such as wormhole routing and virtual channel flow control etc. help keep the extra communication costs to a minimum [BM06a].

7.4.3 Temperature-aware Virtual Submeshes

In addition to the two virtual submeshes shown in Figure 7.8, two more virtual submeshes S_a and S_b are shown as (a) and (b) of Figure 7.9, which can satisfy the request of J as well.

Figure 7.9: Virtual submeshes with different thermal features

However, these virtual submeshes have different thermal features, shown in Table 7.4. For example, among S_a 's cores, the highest temperature is 98.3°C; the lowest temperature is 43.8°C; the average temperature is 68.3°C. Among S_b 's cores, these temperatures are 96.3°C, 41.9°C and 63.8°C respectively. S_b has the best thermal features among these four virtual submeshes and thereby it is preferred for J . Our aim is to find a suitable-sized cool virtual submesh, i.e., a virtual submesh with low average temperature, for an incoming job.

The problem of finding a cool virtual submesh can be declared as follows: for accommodating an $r \times s$ request, to find a submesh with the lowest average temperature from an $m \times n$

Table 7.4: Comparing thermal features of virtual submeshes

Virtual Submesh	Highest Core Temperature	Lowest Core Temperature	Average Core Temperature
Figure 7.8 (a)	98.3	41.9	68.2
Figure 7.8 (b)	98.3	43.8	66.9
Figure 7.9 (a)	98.3	43.8	68.3
Figure 7.9 (b)	96.3	41.9	63.8

submesh of tiles which contains at least $r \times s$ non-overheated tiles and possibly includes some overheated tiles, where $m \geq r$ and $n \geq s$.

Since finding an optimal submesh allocation is NP-complete [LL05], a dynamic programming technique is used to construct a cool virtual submesh in steps to reduce complexity. We will present two such algorithms in the following subsections.

By identifying and allocating cool virtual submeshes to incoming applications, the performance of a multi-/many-core processor is managed in order to achieve improved heat balance and enhanced stability of the chip. If necessary, overheated cores are left out of computation to cool down till they become suitable for re-use.

NoC techniques such as wormhole routing and virtual channel flow control etc. are deployed to keep the extra communication costs caused by virtual submeshes to a minimum [BM06a]. Transient temperatures of cores are inputs to our scheme. In [CS06a], transient temperatures of functional units can be calculated with small overheads with a linear formula based on temperature and activity data. Since various performance counters continue to exist in multi-/many-core chips [ABC⁺06], the methodology in [CS06a] is envisaged to provide transient temperatures of cores after adjustments.

7.4.4 Algorithm “Maximum-Cut”

In this subsection, we present an algorithm for constructing suitable-sized cool virtual submeshes. In this algorithm, we first construct the virtual submesh of the maximum size and then select a suitable-sized virtual submesh at one of its corners. In detail, a cool virtual submesh (CVS) with maximum dimension is firstly constructed by selecting non-overheated tiles. Then a suitable-sized CVS containing required number of non-overheated tiles is “cut” at one corner of this maximum CVS and a physical submesh containing this suitable-sized CVS is allocated to the job. Hereinafter, this algorithm is called “maximum-cut” and is referred to as $Algo_{max}$. The scheme is regarded as *contiguous* since partition for the job, i.e., a physical submesh with

$r \times s$ non-overheated tiles and maybe some overheated tiles, is contiguous. With the help of Figure 7.9, we describe this algorithm in detail as follows.

Firstly, we adopt algorithm *GCR* in [Low00] to construct a common virtual submesh. Request orientation rotation is supported by our scheme, i.e., the method of constructing virtual columns also applies to virtual rows. We describe our scheme in column manner. For simplicity, we assume all rows participate in allocation. Each row contributes one core to a virtual column and each virtual column begins at the top row, R_1 . A core u can also be indicated by (i, j) where i is its row index and j is its column index, $1 \leq i, j \leq 5$. A inner core (i, j) has three successors $(i + 1, j - 1)$, $(i + 1, j)$ and $(i + 1, j + 1)$. Cores on borders have two successors. The set of successors of core u is denoted as $Adj(u)$.

GCR begins by selecting the leftmost non-overheated core, say u , of row R_1 into a virtual column. Next, u 's leftmost successor, say v , is connected to u . This process is repeated as follows: In each step, GCR attempts to connect current core v to its leftmost successor that has not been examined. If GCR fails in doing so, no virtual column that contains v can be formed. Thus, GCR backtracks to the previous core, say w , which was connected to v and attempts to connect w to w 's leftmost successor (excluding v) that has not been examined. This process is repeated until either i) a core v in R_5 is connected to a core in R_4 or ii) GCR backtracks to u in R_1 . For termination under condition i), a virtual column that passes through each of the rows would have been constructed. For termination under condition ii), no virtual column that begins with u can be formed. In the next iteration, GCR attempts to construct a new virtual column by selecting the leftmost core in R_1 that has not been examined and the entire process described above is repeated. GCR finally terminates when all cores in R_1 have been examined. Upon termination of GCR, the cores in each virtual column can be obtained by traversing backward via the chain of predecessors. Three virtual columns constructed in order are shown as C_1 , C_2 and C_3 in Figure 7.9 (a). If submeshes by GCR can not satisfy the request of J even after switching request orientation, our scheme keeps J waiting.

Secondly, we construct a cool virtual submesh based on the virtual submesh constructed by GCR if it satisfies J 's request. $A[B_l, B_r)$ is used to indicate an "area" consisting of cores bounded by B_l and B_r (including B_l but not including B_r) where B_l and B_r are two virtual columns. B_l and B_r are called the left boundary and the right boundary of the area respectively. A virtual column is called a *local optimal cool virtual column* in $A[B_l, B_r)$ if it has the lowest

average temperature of cores among virtual columns in $A[B_l, B_r)$. We dynamically construct such areas to search a local optimal cool virtual column in each area one by one from right to left.

Figure 7.10: Construction of a local optimal cool virtual column

We model this search problem as finding the shortest weighted paths of a directed network $G = (V, E)$. V is the set of all cores; E is the set of directed edges from cores to their successors. A directed edge from core u to core v is denoted as $e(u, v)$ and $cost(u, v)$ indicates the cost of $e(u, v)$ which is defined as the temperature of u , i.e., $temp(u)$. The temperature of an overheated core is defined as ∞ . The cost of a path is defined as the sum of costs of all edges in the path. Cost of the shortest path from u to R_i , say $cost(u, R_i)$, where u is a vertex and R_i is a row, is recursively defined as follows: For any $u \in A[B_l, B_r)$, $cost(u, R_i)$ equals: i) 0, if u is in row R_i . ii) $\min_{v \in \Gamma} \{cost(u, v) + cost(v, R_i)\}$, if the row index of u is smaller than i , $1 \leq i \leq 5$. Here, v is a core and $\Gamma = Adj(u) \cap A[B_l, B_r)$. If Γ is empty, $cost(u, R_i)$ is defined as ∞ . Therefore, a local optimal cool virtual column in $A[B_l, B_r)$ is the shortest path in $A[B_l, B_r)$ from a core in R_1 to R_5 with the minimal cost $\min_{u \in R_1} cost(u, R_5)$. This minimal cost is denoted as $cost(R_1, R_5)$.

We developed an algorithm, namely TA_Local_Opt_Col, to find a local optimal cool virtual column in a bounded area using a dynamic programming approach. Its steps are briefly described as follows:

- (i) Initialize $cost(v, v)$ as $temp(v)$ for each $v \in \{R_5\} \cap A[B_l, B_r)$.

- (ii) Calculate $cost(u, R_i)$ for each $u \in \{R_{i-1}\} \cap A[B_l, B_r)$, for $i = 4, \dots, 1$.
- (iii) Select $\min_{u \in R_1} cost(u, R_5)$, i.e. $cost(R_1, R_5)$, as the cost of the shortest path. If there are multiple paths having this minimal value, choose the rightmost one.
- (iv) Output the calculated shortest path in the trace-back process. In the trace-back process, the rightmost shortest path is selected to save cores for constructing the next local optimal cool virtual column.

In our example, to find the rightmost local optimal cool virtual column, $A[C_3, C_{NULL})$ is constructed and C_1^* is found, shown in Figure 7.10. Here, column C_{NULL} is an imaginary column introduced to facilitate the implementation of the algorithm. In a similar manner, C_2^* is constructed in $A[C_2, C_1^*)$. With C_2^* , C_3^* can be constructed in $A[C_1, C_2^*)$. Finally, combining these local optimal columns forms a cool virtual submesh (Figure 7.9 (b)), from which a suitable-sized submesh with the lowest average temperature is allocated for J at one of its “corners”.

7.4.5 Algorithm “Minimum-Expand”

Further study of the above algorithm “Maximum-cut” reveals that $Algo_{max}$ could miss some good candidates of cool virtual submeshes. This can be explained as follows. $Algo_{max}$ only considers virtual submeshes at corners of maximum CVSes and could miss better candidates located inside them. In addition, it has two forms of fragmentation. On the one hand, inside a partition, when non-overheated tiles outnumber the size of request, this results in “internal fragmentation” which could deteriorate when overheated tiles inside the partition cool down during execution. Techniques such as thermal-aware task migration and load balancing may relieve internal fragmentation but their effects may be limited if too many non-used tiles are there. On the other hand, among partitions, “external fragmentation” may exist when a sufficient number of tiles are available to satisfy a request, but they can not be allocated as a (virtual) submesh of proper shape.

We propose another algorithm for constructing CVSes aiming at overcoming above weaknesses of $Algo_{max}$. The new algorithm starts with finding a suitably-sized cool physical submesh which has the lowest average temperature of tiles. If it fails to accommodate a request

due to overheated tile(s) inside it, this physical submesh is expanded into a bigger one by “annexing” neighboring row(s) or column(s) and a suitable-sized cool virtual submesh is searched using $Algo_{max}$ on the expanded physical submesh(es). This new algorithm is named $Algo_{min}$. With it, a partition containing the suitably-sized virtual submesh is expanded from a physical submesh which could be a “minimum” partition if it has no overheated cores.

Expected benefits of $Algo_{min}$ are as follows.

- (i) It locates cool virtual submeshes more flexibly than $Algo_{max}$.
- (ii) It could help relieving internal fragmentation by replacing or complementing $Algo_{max}$.
- (iii) It paves the way for temperature-aware non-contiguous virtual submesh allocation schemes which usually have less external fragmentation. A partition in the expected non-contiguous schemes can be formed by combining disjoint virtual submeshes constructed by $Algo_{min}$.

Below, we give an illustrative and informal description of $Algo_{min}$. We follow the same assumptions in the previous section. Techniques of NoC such as wormhole routing and virtual channel flow control are deployed to keep extra communication costs at a minimum [BM06a]. With various performance counters in manycore chips [ABC⁺06], the methodology in [CS06a] is envisaged to provide transient temperatures of tiles after appropriate adjustments.

First, several notations are introduced. A tile located at the i th row and the j th column is denoted as $e_{i,j}$. An $r \times s$ physical submesh with $e_{i,j}$ at its top-left corner is denoted as $P_{i,j}^{r \times s}$, where r (s) is the number of rows (columns) of $P_{i,j}^{r \times s}$. The set of x tiles on the i th row (j th column) from $e_{i,j}$ to $e_{i,j+x}$ ($e_{i+x,j}$) is denoted as $P_{i,j}^{1 \times x}$ ($P_{i,j}^{x \times 1}$). The temperature of a set of tiles U is defined as the average temperature over all its elements, denoted as $T(U)$. For example, submesh A^* , i.e., $P_{1,3}^{8 \times 1}$ in the left part of Figure 7.11, is an 8×1 submesh starting from the tile $e_{1,3}$. Its temperature $T(P_{1,3}^{8 \times 1})$ is 65.8°C .

To find a cool submesh $P_{i,j}^{r \times s}$, $Algo_{min}$ scans the whole chip starting from $e_{1,1}$ (the top-left corner) in the order of left-to-right within a row and top-down between rows. The threshold temperature for overheated tiles is assumed to be 100°C . During this process, the temperature of a physical submesh is re-used to calculate the temperature of its neighboring submesh. Based on $P_{i,j}^{r \times s}$, $P_{i,j+1}^{r \times s}$ can be formed by replacing column $P_{i,j}^{r \times 1}$ with column $P_{i,j+s}^{r \times 1}$. Thus, $Algo_{min}$ is able to find such a cool submesh in linear time. In the left part of Figure 7.11, submesh C^* , i.e., $P_{6,5}^{3 \times 4}$, is found for a request of size 3×4 .

 Figure 7.11: Cool virtual submeshes constructed with $Algo_{min}$ (left) and $Algo_{max}$ (right)

However, submesh C^* contains an overheated tile and it has to be expanded to accommodate the 3×4 request. If virtual submeshes are constructed in the column (row) manner, physical submeshes can be expanded by taking in neighboring physical columns (rows). After annexing each column (row), algorithm $Algo_{max}$ is invoked to construct a CVS on the new submesh. This process repeats and then stops when a suitable-sized CVS is found or none can be found even after taking columns (rows) on the borders. In Figure 7.11, virtual columns are constructed and C^* can only take in columns on the left. A CVS having 4 virtual columns (tiles connected with solid lines) is constructed by invoking $Algo_{max}$ after C^* is expanded into B^* . In B^* , tiles connected with dotted lines indicate common columns used by $Algo_{max}$.

The steps of $Algo_{min}$ are listed as follows.

- (i) For an $r \times s$ request R , $Algo_{min}$ searches and finds the coolest $r \times s$ physical submesh P . If P can accommodate R , it is allocated and $Algo_{min}$ stops.
- (ii) Otherwise, P is expanded by annexing n neighboring row(s) or column(s) if possible ($n = 1, 2, \dots$). An $r \times s$ CVS is searched using $Algo_{max}$ on the expanded submesh(es) (there are at most four expanded submeshes based on the $r \times s$ submesh P).
- (iii) If it fails, step 2 repeats after increasing n by 1 until such an $r \times s$ CVS is found or all trials fail.

Table 7.5: Comparison of allocated submeshes by $Algo_{min}$ and $Algo_{max}$

Request	Using $Algo_{min}$			Using $Algo_{max}$		
	A.T.	I.F.R.	P.S.	A.T.	I.F.R.	P.S.
8×1	65.8°C	0%	$P_{1,3}^{8 \times 1}$	58.0°C	66.7%	$P_{1,6}^{8 \times 3}$
3×4	62.8°C	20%	$P_{6,4}^{3 \times 5}$	60.0°C	33.3%	$P_{6,3}^{3 \times 6}$

A.T.: Average Temperature of Tiles in Cool Virtual Submesh

I.F.R.: Internal Fragmentation Ratio

P.S.: Physical Submesh

7.4.6 Comparison of Algorithms

Now, our temperature-aware virtual submesh allocation scheme has two alternative algorithms for submesh allocation, i.e., $Algo_{min}$ and $Algo_{max}$, to construct a CVS for a request. For requests of size 8×1 and 3×4 , several CVSes constructed using these alternatives are shown in the left and right parts of Figure 7.11 respectively. The comparison of these CVSes for these two requests is listed in Table 7.5.

From the results in the table, $Algo_{min}$ can be more flexible than $Algo_{max}$ for some cases. The 8×1 request is a good example for that. A^* found by $Algo_{min}$ is located at central part of the chip compared to A found by $Algo_{max}$. In both cases, CVSes by $Algo_{min}$ have less internal fragmentation than ones by $Algo_{max}$. On the other hand, both CVSes found by $Algo_{min}$ have higher temperature than those found by $Algo_{max}$. It is clear that a trade-off is made between fragmentation and lower temperature.

For $Algo_{max}$, we do brief algorithmic complexity analysis as follows. Assume n and m are the width and height of the NoC mesh respectively. Also assume k independent logical columns have been produced by GCR and σ_{i+1} is the number of the non-overheated cores in the area $A[C_{k-i}, C_i^*)$ where $1 \leq i \leq k$. The calculations for C_{i+1}^* take $O(\sigma_{i+1})$ times because procedure $TA_Local_Opt_Col$ scans σ_{i+1} cores at most four times. So, the time complexity of $Algo_{max}$ is $\sum_{i=0}^{k-1} \sigma_{i+1}$. As the submesh requested by $TA_Local_Opt_Col$ to calculate C_{i+1}^* is of size $m \times k$, the number of the remaining non-overheated cores is $N - m \times k$, where N is the number of the non-overheated cores of the whole NoC mesh. The worst case of $Algo_{max}$ is that all $N - m \times k$ remains are located in the area $A[C_{k-i}, C_i^*)$. Noting that

$$m \times k \leq N \leq m \times n \text{ and } m \leq \sigma_{i+1} \leq m + (N - m \times k)$$

we obtain:

$$m \times k \leq \sum_{i=0}^{k-1} \sigma_{i+1} \leq m \times k \times (1 + n - k) \leq N \times (1 + n - k)$$

Therefore, the worst time complexity of $Algo_{max}$ is bounded by $O(N \times (n - k))$.

For $algo_{min}$, except the case that the expanded submesh by $Algo_{min}$ is the whole NoC, the expanded submeshes used in $Algo_{min}$ can be covered by the submeshes similar to those used in $Algo_{max}$. Thus, the number of cores scanned by procedure TA_Local_Opt_Col is smaller and the worst time complexity of $Algo_{min}$ can also be bounded by $O(N \times (n - k))$.

7.5 An Adaptive Scheme for Temperature-aware Submesh Allocation

In this section, we introduce an adaptive scheme for temperature-aware submesh allocation. The allocation algorithm of this adaptive scheme is shown in Figure 7.12. This scheme is adaptive to the existence of overheated cores in the system when the system tries to allocate resources for an incoming application. When there are no overheated cores in the system, this adaptive scheme takes the path from (1) to (3) in Figure 7.12, working in the exactly same way as the submesh allocation scheme described in the preceding chapter. When there are overheated cores in the system, this adaptive scheme takes the path from (2) to (3) in Figure 7.12. This adaptive scheme uses the same data structure, i.e., free list of free submeshes which is described in the preceding chapter, to organize resources.

We focus on the submesh allocation of the adaptive scheme when there are overheated cores in the system. For all the free submeshes in the free list (introduced in the preceding chapter), the scheme checks them one by one until suitable resources for the application are identified.

The algorithm gets an allocation submesh from the free list and then counts the non-overheated cores within it. If the number of non-overheated cores within the submesh is not enough to satisfy the request of the incoming application, the current submesh is skipped and the algorithm will consider the next submesh from the free list.

If the number of non-overheated cores within the submesh is enough to satisfy the request and none of these cores is overheated, a candidate submesh is determined for the application.

If the number of non-overheated cores within the submesh is enough to satisfy the request and some of these cores are overheated, the algorithms “Maximum-cut” and “Minimum-expand” are called to construct two virtual submeshes if possible.

If none of the constructed virtual submeshes can satisfy the request, the current submesh is skipped and the algorithm will consider the next submesh from the free list. If both of the

ical contiguous submesh which covers the virtual submesh and allocates it to the request. On such a physical contiguous submesh, there are several extra cores, which could be overheated or non-overheated. These extra cores can be utilized in runtime thermal management as described below.

After a physical contiguous submesh is identified for the application, the algorithm does the housekeeping work to maintain the free list of free submeshes.

There is also the possibility that no physical contiguous submesh can be identified for the incoming application. When this happens, the application has to wait for future allocation when some resources are released.

7.6 Runtime Thermal Management for NoC based Many-core Systems

The adaptive scheme for temperature-aware submesh allocation described in above section tries to identify thermally favourable resources for an incoming application when it arrives at the system. The temperature-aware resource allocation is a one time effort and provides a good starting point for thermal management. In this section, we introduce a technique for runtime thermal management for NoC based manycore systems after applications start running on their respective allocated resources. The technique is described by the flow chart in Figure 7.13.

An assumption for the runtime thermal management is that the system carries out the thermal management at fixed intervals, starting with checking and updating temperature of compute cores, shown in Figure 7.13. An interval does not need to be very small since the temperatures of cores change slowly. Hence, an interval can be 100,000 cycles.

The system configuration is considered as changed when the following two categories of *resource change events* happen: 1) The temperatures of some cool cores become higher than the “overheated” threshold value and become overheated due to the power consumption for computation; 2) The temperatures of some overheated cores become lower than the “cool” threshold value and become suitable for computation again. When there are changes in system configuration, the system executes the runtime thermal management immediately. Otherwise, the runtime thermal management is carried out at the next interval.

When the changes in the system configuration are detected, partitions where resource change events occur are marked as changed. After the marking is completed, the system handles the changed partitions one by one.

Figure 7.13: A technique for runtime thermal management

If a changed partition does not contain a virtual submesh, according to the above adaptive scheme for submesh allocation, all cores within this partition are participating in the computa-

tion and the resource change events mean that some cores must have become overheated after a period of computation. For this situation, a new submesh which has enough cool cores should be provided to the application of the partition. To identify the new submesh, the adaptive scheme for submesh allocation can be executed. If a suitable submesh can be found, the application can be migrated to the new submesh. If no suitable submesh can be found, the application has to be suspended in order to safeguard the processor and it can be restarted later.

If a changed partition does contain a virtual submesh, there should be several redundant cores within this partition. The resource change events for this partition can be both categories. To better utilize the cool cores, the system calls the “Maximum-cut” and “Minimum-expand” algorithms aiming to construct thermally favourable virtual submeshes, i.e., cool virtual submeshes, on the partition. If a new cool virtual submesh can be constructed on the current changed partition, the tasks of the application of the partition can be rearranged onto the new virtual submesh by necessarily migrating some tasks. If no new cool virtual submesh can be found on the current changed partition, a new submesh which has enough cool cores should be provided to the application. For this purpose, the adaptive scheme for submesh allocation should be executed, as described above.

After all changed partitions are handled, the runtime thermal management for the current interval ends. At the next interval, the runtime thermal management starts again.

7.7 Summary

This chapter first presents a temperature-aware contiguous submesh allocation scheme for future multi-/many-core chips which helps to balance heat chip-wide and hence improves the device lifetime and reliability.

Then, we introduce a new form of virtual submesh and temperature-aware virtual submeshes for handling overheated cores on NoC based manycore systems. Two fast algorithms have been developed for constructing temperature-aware virtual submeshes for incoming applications. Based on the above, an adaptive scheme for temperature-aware submesh allocation is described to flexibly handle different resource allocation situations under the existence of overheated cores in the system.

To complement the adaptive scheme for temperature-aware submesh allocation, a runtime thermal management technique is presented to further balance the heat during executions of

applications. In this way, a complete thermal management solution is formed for NoC based manycore systems which adopt submesh based resource management.

7.7.1 Novelty of Our Research

First, existing work focused on the multicore systems with a small number of cores and suitable thermal management techniques for NoC based manycore systems with a large amount of cores are absent in the literature. To the best of knowledge, we are the first to explore the thermal management problems on manycore systems by introducing temperature-aware submeshes.

Second, unlike the existing work which only adopts a reactive strategy for eliminating thermal emergencies through monitoring and control, the novelty of our work on runtime thermal management lies in that 1) it first adopts a proactive strategy of preventing thermal emergencies by choosing resources in a temperature-aware manner, i.e., choosing thermally favourable resources, for incoming applications at resource allocation stage; 2) then it adopts the reactive strategy via monitoring and task migration for eliminating thermal emergencies.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

A number of novel techniques have been proposed in this thesis for adopting manycore systems in emerging compute intensive applications. A cycle-level simulation framework for NoC based manycores has been devised. The underlying modular architecture of the proposed framework lends well for scalability and is capable of supporting different topologies and various configurable parameters. A simulator for NoC-based homogeneous embedded manycore systems has been implemented and tested with the help of PowerPC 405 cores. New instructions for interfacing the NoC, an MPI compatible library and GNU based cross-compiler tool-chain have been devised and incorporated into the simulator to support the mapping of parallel applications that communicate via on-chip network. A detailed evaluation of an MPI based program shows that the simulator is capable of capturing accurate performance data during simulation runs. The simulator was implemented based on the proposed framework in order to fully represent the physical system in terms of functionality and complexity. Debugging tools for hardware and applications were relied upon to eliminate memory leakages within the simulation environment. Libraries that support NoC communications and the MPI programming model have been developed from scratch and the logging systems have been implemented to fully test and verify the proposed simulator.

An efficient approach to accelerate the cycle-level micro-architectural simulations of many-core systems was proposed and implemented. It relies on transforming the original single-threaded UNISIM engine into a multithreaded engine so as to leverage on the multicore based platforms available in the market today. A systematic approach to facilitate the exploitation of the fine-grained parallelism within simulated cycles has been introduced. This provides for

dividing simulation computation workload into a number of partitions to support concurrent execution. The technique for overcoming unacceptable computation variations through dynamic balancing of the workloads of partitions leads to notable performance improvements. Moreover, provisions have been made to facilitate user managed adaptive simulations on a case-by-case basis. Experiments on an 8-core computer show that speedups of close to 6X can be achieved to accelerate the proposed simulator for manycore systems. It appears that the worker threads contribute significantly towards notable acceleration during adaptive simulations. While the proposed cycle-level micro-architectural simulator for the NoC based manycore systems lends well for mapping onto multi-cores with increased number of cores on the simulated systems, we observe that good speedups were difficult to achieve for small sized problems that have been investigated. Also, it is evident that the sequential computations of the multithreaded simulations must be kept to a minimum, thereby leveraging on the parallel computations to achieve high speedups.

A runtime resource management method that relies on submesh based allocations has been devised and tested for resource management on NoC based manycores. In particular, the proposed strategy employs a hierarchical scheme to facilitate a highly scalable runtime resource management on NoC based embedded manycores. Unlike existing strategies that aim to obtain the highest utilization of resources at the expense of increasing the complexity of allocation algorithms and communication contentions, the proposed strategy relies on the notion that there can be redundant cores that need not be powered on simultaneously. In the proposed approach, each application is assigned with a submesh in order to reduce the complexity of the allocation algorithm and to remove the external communication contentions. The redundant cores within a submesh are put to sleep during execution, thereby satisfying the need to keep the power consumption to a minimum. The proposed hierarchical scheme allocates submeshes at the global level and relies on a local manager for allocation and de-allocation within each submesh. This helps to improve performance and to minimize the communication costs especially during resource de-allocations. The approach taken to trade reduced utilization of cores for lowering complexity of resource management has shown to reap notable benefits for the case of NoC based manycores.

The technique proposed for hybrid non-preemptive/cooperative multi-tasking has been shown to overcome the limitations of non-preemptive multi-tasking, which is widely adopted by existing strategies. Architectural support and the method for parallelizing applications have been

proposed to implement the hybrid multi-tasking. An MPEG-2 encoder code was parallelized for evaluation using the hybrid multi-tasking technique to demonstrate that the parallelized MPEG-2 encoder implementation scales well with the number of compute resources. Moreover, the hybrid multi-tasking has been shown to be capable of adapting execution allocations dynamically based on the availability of resources.

A runtime thermal management technique on NoC based manycore systems has been devised to achieve heat balance in a proactive way via a thermal-aware contiguous submesh allocation process. The allocation policies have been amended to consider temperatures of cores in an attempt to minimize the occurrence of overheated cores. Experiments show that the proposed thermal-aware scheme can reduce both the temporal variance and spatial variance of temperature of cores. The approach proposed to treat overheated cores as temporarily faulty utilizes virtual submesh, which is capable of utilizing more idle cores compared to the virtual submesh schemes proposed in existing work. Two fast heuristics have been devised using dynamic programming to dynamically construct virtual submeshes with thermally favorable resources. A runtime thermal-aware task migration technique has also been proposed to balance heat during the execution of tasks by selecting idling cores (redundant ones or those that have cooled down) by migrating tasks to more favorable cores. This together with an adaptive scheme to provide thermal-aware contiguous or virtual submeshes for incoming applications leads to a more comprehensive solution for thermal-aware management of NoC based manycores.

8.2 Future Work

This research can be extended in many ways. As an immediate step, the simulation framework can be further extended to support other NoC based multi-/many-core systems. The proposed simulator framework can be relied upon for rapid realization of NoC based heterogeneous manycore chips supporting shared memory model based on processor models such as PPE and SPE of the CellSim simulator [Cel09]. The major work in realizing such a simulator will involve the replacement of the ring based EIB bus of CellBE with an NoC. Moreover, improving usability of our proposed simulation environment by accommodating a variety of processing components such as processor cores and NoC structures will be of interest to the research community.

It will be of interest to explore ways to further accelerate the proposed cycle-level micro-architectural simulations. At present, the proposed acceleration technique is not well suited for systems with small simulation computation in each simulated cycle. This requires further investigation. Methods for accelerating simulations with parallel hardware devices could lead to interesting outcomes. For example, methods to accelerate SystemC cycle-level simulation with GPUs could provide for opportunities to deploy alternate compute platforms.

Further investigations to determining the optimal submesh for an application at runtime will be of interest. By extending the work in this thesis, several candidate submeshes can be prepared for an application in preprocessing and OS can choose the most suitable one based on the runtime situation in order to improve flexibility, turn-around time and other performance metrics. This requires the exploration of fast heuristics for evaluating runtime situations.

At present, applications that have data parallelism have been successfully parallelized to support the hybrid non-preemptive/cooperative multi-tasking. Parallelizing applications, which exhibit alternate categories of parallelism, for hybrid multi-tasking is worth exploring. This should lead to wider application of the hybrid multi-tasking.

As fine-grained power management techniques are emerging in manycore systems, the models of processor cores should be variable as functional blocks of cores could be turned off/on at runtime. Hence, accurate and flexible models for processor cores will lead to improvements to the thermal-aware management of resources. However, this must be balanced with the increased simulation time of processor cores.

Run-time techniques for dynamic thermal management can be further improved by devising efficient on-line monitoring techniques that can help to establish the most appropriate point for invoking the thermal-aware migration of threads such that system-wide temperature variations and costs of migration can be optimally managed.

Implementing a small-scale version of the proposed NoC based manycore system on FPGA will be of interest. This should pave the way for evaluating the proposed techniques using real-life applications. The OpenSPARC T1 design could be a good starting point to investigate if it will also lead to notable speed-up over software-based approach. If successful, it can help accelerate future research efforts in NoC based manycore systems.

References

- [Aba06] I. Ababneh. An Efficient Free-List Submesh Allocation Scheme for Two-Dimensional Mesh-Connected Multicomputers. *Journal of Systems and Software*, 79(8):1168–1179, 2006.
- [ABC⁺06] 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 UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 18, 2006.
- [ACG⁺07] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *Computer Architecture Letters*, 6(2):45–48, Jul.-Dec. 2007.
- [ACJ⁺] Mani Azimi, Naveen Cherukuri, D. N. Jayasimha, Akhilesh Kumar, Partha Kundu, Seungjoon Park, Ioannis Schoinas, and Aniruddha S. Vaidya. Integration Challenges and Tradeoffs for Tera-Scale Architectures. *Intel Technology Journal*, <http://www.intel.com/technology/itj/2007/v11i3/1-integration>.
- [ACPP08] Giuseppe Ascia, Vincenzo Catania, Maurizio Palesi, and Davide Patti. Implementation and Analysis of A New Selection Strategy for Adaptive Routing in Networks-on-Chip. *Computers, IEEE Transactions on*, 57(6):809–820, June 2008.

REFERENCES

- [AHKB00] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate versus IPC: The End of The Road for Conventional Microarchitectures. *SIGARCH Comput. Archit. News*, 28(2):248–259, 2000.
- [APJ08] Niket Agarwal, Li-Shiuan Peh, and Niraj Jha. GARNET: A Detailed Interconnection Network Model inside A Full-System Simulation Framework. Technical Report CE-P08-001, Princeton University, 2008.
- [ASU10] Arizona State University. Forman YUV Video Sequences. <http://trace.eas.asu.edu/yuv/index.html>, 2010.
- [BA97] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1997.
- [BABP06] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 15–20, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.
- [BBD⁺08] Michael A. Bender, David P. Bunde, Erik D. Demaine, Sandor P. Fekete, Vitus J. Leung, Henk Meijer, and Cynthia A. Phillips. Communication-Aware Processor Allocation for Supercomputers: Finding Point Sets of Small Average Distance. *Algorithmica*, 50(2):279–298, 2008.
- [BD02] L. Benini and G. DeMicheli. Networks on Chips: A New Paradigm for Component-Based MPSoC Design. In *L. Benini and G. DeMicheli, Networks on Chips: A New Paradigm for Component-Based MPSoC Design, IEEE Computer, Jan. 2002, pp. 70-78.*, 2002.
- [BD06] James Balfour and William J. Dally. Design Tradeoffs for Tiled CMP On-Chip Networks. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 187–198, New York, NY, USA, 2006.

REFERENCES

- [BDH⁺06] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The M5 Simulator: Modeling Networked Systems. In *IEEE Micro*, 26(4):52 – 60, 2006.
- [BM03] P. Bhojwani and R. Mahapatra. Interfacing Cores with On-Chip Packet-Switched Networks. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 382–387, Jan. 2003.
- [BM06a] Luca Benini and Giovanni De Micheli. *Networks on Chips: Technology and Tools*. Morgan Kaufmann Publishers, 2006.
- [BM06b] Praveen Bhojwani and Rabi N. Mahapatra. Core Network Interface Architecture and Latency Constrained On-Chip Communication. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 358–363, Washington, DC, USA, 2006.
- [BM06c] Tobias Bjerregaard and Shankar Mahadevan. A Survey of Research and Practices of Network-on-Chip. *ACM Comput. Surv.*, 38(1):1, 2006.
- [BM09] Paul Bogdan and Radu Marculescu. Statistical Physics Approaches for Network-on-Chip Traffic Characterization. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 461–470, New York, NY, USA, 2009.
- [BMOKA07] S. Bani-Mohammad, M. Ould-Khaouaa, and I. Ababneh. An Efficient Non-Contiguous Processor Allocation Strategy for 2D Mesh Connected Multicomputers. *Information Sciences*, 177(14), pages 2867-2883, 2007.
- [BMOKAM07] Saad Bani-Mohammad, Mohamed Ould-Khaoua, Ismail Ababneh, and Lewis M. Mackenzie. An Efficient Processor Allocation Strategy That Maintains A High Degree of Contiguity Among Processors in 2D Mesh Connected Multicomputers. In *In proceedings of IEEE/ACS International Conference Computer Systems and Applications(AICCSA '07)*, 2007.
- [Bor05] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *Micro, IEEE*, 25(6):10 – 16, Nov.-Dec. 2005.

REFERENCES

- [Bor07] Shekhar Borkar. Thousand Core Chips: A Technology Perspective. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 746–749, New York, NY, USA, 2007.
- [BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of 27th International Symposium on Computer Architecture*, pages 83 - 94, 2000.
- [BV06] E. Beigne and P. Vivet. Design of On-Chip and Off-Chip Interfaces for A GALS NoC Architecture. In *ASYNC '06: Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, page 172-181, Washington, DC, USA, 2006.
- [BVC04] Nilanjan Banerjee, Praveen Vellanki, and Karam S. Chatha. A Power and Performance Model for Network-on-Chip Architectures. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21250, Washington, DC, USA, 2004.
- [BYM⁺07] B. Baas, Zhiyi Yu, M. Meeuwsen, O. Sattari, R. Apperson, E. Work, J. Webb, M. Lai, T. Mohsenin, D. Truong, and J. Cheung. ASAP: A Fine-Grained Many-Core Platform for DSP Applications. *Micro, IEEE*, 27(2):34 –45, March-April 2007.
- [Car01] D. Carmean. The Pentium 4 Processor. In *Hot Chips 13*, 2001.
- [CBC⁺10] Juan A. Colmenares, Sarah Bird, Henry Cook, Paul Pearce, David Zhu, John Shalfy, Steven Hofmeyry, Krste Asanovic, and John Kubiawicz. Resource Management in the Tessellation Manycore OS. *Proceedings of the Second USENIX conference on Hot topics in parallelism (HotPar'10)*, 2010.
- [CC99] Ge-Ming Chiu and Shin-Kung Chen. An Efficient Submesh Allocation Scheme for Two-Dimensional Meshes with Little Overhead. *IEEE Trans. on Parallel and Distributed Systems*, 10(5), pages 471 - 486, 1999.

- [CCD⁺08] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory. Relaxing Synchronization in A Parallel Systemc Kernel. In *Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on*, pages 180–187, Dec. 2008.
- [CCM07] E. Carvalho, N. Calazans, and F. Moraes. Heuristics for Dynamic Task Mapping in NoC-Based Heterogeneous MPSoCs. In *Rapid System Prototyping, 2007. RSP 2007. 18th IEEE/IFIP International Workshop on*, pages 34 –40, 28-30 2007.
- [Cel09] Cellsim. Cellsim Simulator, <http://pcsostres.ac.upc.edu/cellsim/doku.php/start>, 2009.
- [Che06] Tze-Chiang Chen. Where Is Cmos Going: Trendy Hype Versus Real Technology. Keynote, The International Solid State Circuits Conference, 2006, 2006.
- [Chi00] Ge-Ming Chiu. The Odd-Even Turn Model for Adaptive Routing. *Parallel and Distributed Systems, IEEE Transactions on*, 11(7):729–738, Jul 2000.
- [CM07] Chen-Ling Chou and Radu Marculescu. Incremental Run-Time Application Mapping for Homogeneous NoCs with Multiple Voltage Levels. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 161–166, New York, NY, USA, 2007.
- [CM08] Chen-Ling Chou and R. Marculescu. User-Aware Dynamic Task Allocation in Networks-on-Chip. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 1232 –1237, March 2008.
- [CM10] Chen-Ling Chou and Radu Marculescu. Run-Time Task Allocation Considering User Behavior in Embedded Multiprocessor Networks-on-Chip. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 29(1), pages 78 - 91, 2010.

REFERENCES

- [COM08] Chen-Ling Chou, U.Y. Ogras, and R. Marculescu. Energy- and Performance-Aware Incremental Mapping for Networks on Chip with Multiple Voltage Levels. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1866–1879, Oct. 2008.
- [CP03] Xuning Chen and Li-Shiuan Peh. Leakage Power Modeling and Optimization in Interconnection Networks. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 90 - 95, 2003.
- [CS06a] S. Chung and K. Skadron. Using On-Chip Event Counters for High-Resolution, Real-Time Temperature Measurement. In *Proceedings. 10th ICTTPES*, 2006.
- [CS06b] S.W. Chung and K. Skadron. A Novel Software Solution for Localized Thermal Problems. In *Technical Report CS-2006-10, Univ. of Virginia Dept. of CS*, 2006.
- [CT94] Po-Jen Chuang and Nian-Feng Tzeng. Allocating Precise Submeshes in Mesh Connected Systems. *IEEE Trans. on Parallel and Distributed Systems*, 5(2), pages 211 - 217, 1994.
- [CWS07] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. A Case for An Over-Provisioned Multicore System: Energy Efficient Processing of Multithreaded Programs. Technical report, 2007.
- [Dal90] William J. Dally. Virtual-Channel Flow Control. *SIGARCH Comput. Archit. News*, 18(3a):60–68, 1990.
- [Dal92] William J. Dally. Virtual-channel Flow Control. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2):194 – 205, Mar 1992.
- [DB93] Jianxun Ding and Laxmi N. Bhuyan. An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Systems. In *Proceedings of the International Conference on Parallel Processing*, 2, pages 193 - 200, 1993.
- [Dev04] Advanced Micro Devices. AMD Demonstrates Dual Core Leadership. In <http://www.amd.com/>, 2004.

REFERENCES

- [Dic10] R. Dick. Embedded System Synthesis Benchmarks Suites (E3S)[online], available <http://ziyang.eecs.umich.edu/dickrp/e3s>, 2010.
- [DJR01] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A Multiprocessor SoC for Advanced Set-Top Box and Digital TV Systems. In *IEEE Design and Test of Computers*, 18(5), pages 21 - 31, 2001.
- [DM06a] James Donald and Margaret Martonosi. An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation. In *Computer Architecture Letters*, 5(2), pages 14-17, 2006.
- [DM06b] James Donald and Margaret Martonosi. Techniques for Multicore Thermal Management: Classification and New Exploration. In *Proceedings of the ISCA*, pages 78 - 88, 2006.
- [DS87] W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *Computers, IEEE Transactions on*, 36(5):547 – 553, May 1987.
- [DT01] W.J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Design Automation Conference, 2001. Proceedings*, pages 684 – 689, 2001.
- [DT03] William J. Dally and Brian P. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2003.
- [Dua93] José Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Trans. Parallel Distrib. Syst.*, 4(12):1320–1331, 1993.
- [DYL02] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [EBS⁺04] L. Eeckhout, Jr. Bell, R.H., B. Stougie, K. De Bosschere, and L.K. John. Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 350 – 361, 2004.

REFERENCES

- [EP04] Noel Eisley and Li-Shiuan Peh. High-Level Power Analysis for On-Chip Networks. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 104–115, New York, NY, USA, 2004.
- [FMLD07] J. Flich, A. Mejia, P. Lopez, and J. Duato. Region-Based Routing: An Efficient Routing Mechanism to Tackle Unreliable Hardware in Network on Chips. In *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, pages 183–194, Washington, DC, USA, 2007.
- [GDR05] K. Goossens, J. Dielissen, and A. Radulescu. AEthereal Network on Chip: Concepts, Architectures, and Implementations. *Design Test of Computers, IEEE*, 22(5):414–421, Sept.-Oct. 2005.
- [Gee05] David Geer. Industry Trends: Chip Makers Turn to Multicore Processors. *Computer*, 38(5):11–13, 2005.
- [GG00] Pierre Guerrier and Alain Greiner. A Generic Architecture for On-Chip Packet-Switched Interconnections. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 250–256, New York, NY, USA, 2000.
- [GN92] Christopher J. Glass and Lionel M. Ni. The Turn Model for Adaptive Routing. *SIGARCH Comput. Archit. News*, 20(2):278–287, 1992.
- [GN94] Christopher J. Glass and Lionel M. Ni. The Turn Model for Adaptive Routing. *J. ACM*, 41(5):874–902, 1994.
- [HBHT08] Kai Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele. Scalably Distributed Systemc Simulation for Embedded Applications. In *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, pages 271–274, June 2008.
- [HKPM09] Rickard Holsmark, Shashi Kumar, Maurizio Palesi, and Andres Mejia. HIRA: A Methodology for Deadlock Free Routing in Hierarchical Networks on Chip.

- In *NOCS '09: Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 2–11, Washington, DC, USA, 2009. IEEE Computer Society.
- [HM04] Jingcao Hu and Radu Marculescu. DyAD: Smart Routing for Networks-on-Chip. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 260–263, New York, NY, USA, 2004.
- [HMH01] R. Ho, K.W. Mai, and M.A. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, apr 2001.
- [HNO97] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A Single-Chip Multiprocessor. In *IEEE Computer*, 30(9), pages 79 - 85, 1997.
- [HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann, 2006.
- [HPK08] Rickard Holsmark, Maurizio Palesi, and Shashi Kumar. Deadlock Free Routing Algorithms for Irregular Mesh Topology NoC Systems with Rectangular Regions. *J. Syst. Archit.*, 54(3-4):427–440, 2008.
- [HVS⁺07] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz Mesh Interconnect for A Teraflops Processor. *Micro, IEEE*, 27(5):51–61, Sept.-Oct. 2007.
- [HWC04] Jörg Henkel, Wayne Wolf, and Srimat Chakradhar. On-Chip Networks: A Scalable, Communication-Centric Embedded System Design Paradigm. In *VLSID '04: Proceedings of the 17th International Conference on VLSI Design*, page 845, Washington, DC, USA, 2004.
- [HYX⁺10] Derek Hower, Luke Yen, Min Xu, Milo Martin, Doug Burger, and Mark Hill. Simulators and Tools for Computer Architecture Research. <http://www.cs.wisc.edu/arch/www/tools.html>, 2010.
- [IBM10] IBM. Coreconnect Bus Architecture, https://www-01.ibm.com/chips/techlib/techlib.nsf/products/coreconnect_bus_architecture, 2010.

REFERENCES

- [ITR06] International Technology Roadmap for Semiconductors (ITRS) 2006, <http://www.itrs.net>, 2006.
- [KBD07] John Kim, James Balfour, and William Dally. Flattened Butterfly Topology for On-Chip Networks. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–182, Washington, DC, USA, 2007.
- [KDA07] John Kim, William J. Dally, and Dennis Abts. Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 126–137, New York, NY, USA, 2007.
- [KDSA08] John Kim, William J. Dally, Steve Scott, and Dennis Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 77–88, Washington, DC, USA, 2008.
- [Keg09] Dan Kegel. GCC/Glibc Cross Toolchains. <http://www.kegel.com/crosstool>, 2009.
- [KHH08] Linda Kaouane, Dominique Houzet, and Sylvain Huet. SysCellC: SystemC on Cell. In *ICCSA '08: Proceedings of the 2008 International Conference on Computational Sciences and Its Applications*, pages 234–244, Washington, DC, USA, 2008.
- [KK98] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.
- [KK09a] John Kim and Hanjoon Kim. Router Microarchitecture and Scalability of Ring Topology in On-Chip Networks. In *NoCArc '09: Proceedings of the 2nd International Workshop on Network on Chip Architectures*, pages 5–10, New York, NY, USA, 2009.

- [KKS⁺07] A. Kumary, P. Kunduz, A.P. Singhx, L.-S. Pehy, and N.K. Jhay. A 4.6Tbits/S 3.6Ghz Single-Cycle NoC Router with A Novel Switch Allocator in 65nm CMOS. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 63–70, Oct. 2007.
- [KPKJ07] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express Virtual Channels: Towards The Ideal Interconnection Fabric. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 150–161, New York, NY, USA, 2007.
- [KPT⁺05] Jongman Kim, Dongkook Park, T. Theocharides, N. Vijaykrishnan, and Chita R. Das. A Low Latency Router Supporting Adaptivity for On-Chip Interconnects. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 559–564, New York, NY, USA, 2005.
- [KS04] Tejas S. Karkhanis and James E. Smith. A First-Order Superscalar Processor Model. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 338, Washington, DC, USA, 2004.
- [KST04a] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 Chip: A Dual-Core Multi-threaded Processor. *Micro, IEEE*, 24(2):40 – 47, mar-apr 2004.
- [KY98] Geunmo Kim and Hyunsoo Yoon. On Submesh Allocation for Mesh Multicomputers: A Best-Fit Allocation and A Virtual Submesh Allocation for Faulty Meshes. *Parallel and Distributed Systems, IEEE Transactions on*, 9(2):175–185, Feb 1998.
- [LAS⁺09] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, New York, NY, USA, 2009.
- [LBYB08] Seung Eun Lee, Jun Ho Bahn, Yoon Seok Yang, and Nader Bagherzadeh. A Generic Network Interface Architecture for A Networked Processor Array

- (NEPA). In *ARCS'08: Proceedings of the 21st international conference on Architecture of computing systems*, pages 247–260, Berlin, Heidelberg, 2008. Springer-Verlag.
- [LC91] Keqin Li and Kam-Hoi Cheng. A Two-Dimensional Buddy System for Dynamic Resource Allocation in A Partitionable Mesh Connected System. *Journal of Parallel and Distributed Computing*, 12(1), pages 79-83, 1991.
- [LGXW09] Mingche Lai, Lei Gao, Nong Xiao, and Zhiying Wang. An Accurate and Efficient Performance Analysis Approach Based on Queuing Model for Network on Chip. In *ICCAD '09: Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 563–570, New York, NY, USA, 2009.
- [LHLB95] T. Liu, W.-K. Huang, F. Lombardi, and L.N. Bhuyan. A Submesh Allocation Scheme for Mesh-Connected Multiprocessor Systems. In *In Proceedings of the International Conference on Parallel Processing*, pages 159-163, 1995.
- [LKB⁺09] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyry, Krste Asanovic and John Kubiawicz. Tessellation: space-time partitioning in a manycore client OS. *Proceedings of the First USENIX conference on Hot topics in parallelism (HotPar'09)*, 2009.
- [LL05] Zne-Jung Lee and Chou-Yuan Lee. A Hybrid Search Algorithm with Heuristics for Resource Allocation Problem. *Inf. Sci. Inf. Comput. Sci.*, 173(1-3):155–167, 2005.
- [LMMM08] Jieyi Long, Seda Ogrenci Memik, Gokhan Memik, and Rajarshi Mukherjee. Thermal Monitoring Mechanisms for Chip Multiprocessors. *ACM Trans. Archit. Code Optim.*, 5(2):1–33, 2008.
- [Low00] Chor Ping Low. An Efficient Reconfiguration Algorithm for Degradable VLSI/WSI Arrays. *IEEE Trans. Comput.*, 49(6):553–559, 2000.
- [LSA⁺05] Man-Lap Li, R. Sasanka, S.V. Adve, Yen-Kuang Chen, and E. Debes. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 34 – 45, 2005.

REFERENCES

- [Ltd10] ARM Ltd. Advanced Microcontroller Bus Architecture (AMBA), <http://www.arm.com/products/system-ip/amba/>, 2010.
- [Lu04] Z. Lu. Analysis of Temporal and Spatial Temperature Gradients for IC Reliability. In *Technical Report CS-2004-08, Univ. of Virginia Dept. of CS*, 2004.
- [LWLN97] Virginia Lo, Kurt J. Windisch, Wanqian Liu, and Bill Nitzberg. Noncontiguous Processor Allocation Algorithms for Mesh-Connected Mutlicomputers. *IEEE Trans. on Parallel and Distributed Systems*, 8(7), pages 712 - 726, 1997.
- [Mar03] T. Maruyama. SPARC64 VI: Fujitsu's Next Generation Processor. In *Microprocessor Forum 2003*, 2003.
- [MAS⁺05] Shankar Mahadevan, Federico Angiolini, Michael Storgaard, Rasmus Grondahl Olsen, Jens Sparso, and Jan Madsen. A Network Traffic Generator Model for Fast Network-on-Chip Simulation. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 780–785, Washington, DC, USA, 2005.
- [MB04] C. McNairy and R. Bhatia. Montecito - The Next Product in The Itanium Processor Family. In *Hot Chips 16*, <http://www.hotchips.org/archive/>, 2004.
- [MCE⁺02] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. In *IEEE Journal of Computer*, 35(2), pages 50 - 58, 2002.
- [McK04] Sally A. McKee. Reflections on The Memory Wall. In *CF '04: Proceedings of the 1st conference on Computing Frontiers*, page 162, New York, NY, USA, 2004.
- [Mer05] A. Merkel. Event-Driven Thermal Management in SMP Systems. In *Proceedings of the Second Workshop on Temperature-Aware Computer Systems (TACS'05)*, 2005.
- [Mic07] Michigan. M5 simulator, 2007. <http://www.m5sim.org/>.

REFERENCES

- [MKY⁺09] H. Matsutani, M. Koibuchi, Y. Yamada, D.F. Hsu, and H. Amano. Fat H-Tree: A Cost-Efficient Tree-Based On-Chip Network. *Parallel and Distributed Systems, IEEE Transactions on*, 20(8):1126–1141, Aug. 2009.
- [MMB07] Orlando Moreira, Jacob Jan-David Mol, and Marco Bekooij. Online Resource Management in A Multiprocessor with A Network-on-Chip. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1557–1564, New York, NY, USA, 2007.
- [MMBNBR07] Francisco Javier Mesa-Martinez, Michael Brown, Joseph Nayfach-Battilana, and Jose Renau. Measuring Performance, Power, and Temperature from Real Processors. In *1st Workshop on Experimental Computer Science (FCRC)*, 2007.
- [Moo00] C. Moore. Power4 System Microarchitecture. In *Microprocessor Forum 2000*, 2000.
- [Moo07] Moore’s Law, <http://www.intel.com/technology/mooreslaw/index.htm>, 2007.
- [MPF⁺09] A. Mejia, M. Palesi, J. Flich, S. Kumar, P. Lopez, R. Holsmark, and J. Duato. Region-Based Routing: A Mechanism to Support Efficient Routing Algorithms in NoCs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(3):356–369, march 2009.
- [MPI09] Message Passing Interface Forum, <http://www.mpi-forum.org/>, 2009.
- [MSB⁺05] 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(4):92–99, 2005.
- [MWM04] Robert Mullins, Andrew West, and Simon Moore. Low-Latency Virtual-Channel Routers for On-Chip Networks. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 188, Washington, DC, USA, 2004.

REFERENCES

- [Nar05] S. Narayanan. Temperature-Sensitive Loop Parallelization for Chip Multiprocessors. In *Proceedings of ICCD*, pages 677 - 682, 2005.
- [Nar06] S. Narayanan. Compiler-Directed Power Density Reduction in NoC-Based Multicore Designs. In *Proceedings of 7th ISQED*, pages 575 - 580, 2006.
- [NC09] Nikita Nikitin and Jordi Cortadella. A Performance Analytical Model for Network-on-Chip with Constant Service Time Routers. In *ICCAD '09: Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 571–578, New York, NY, USA, 2009.
- [NG07] Youssef N. Naguib and Rafik S. Guindi. Speeding Up SystemC Simulation Through Process Splitting. In *DATE '07: Proceedings of the conference on Design, Automation and Test in Europe*, pages 111–116, San Jose, CA, USA, 2007.
- [Nir07] NIRGAM NoC Simulator, <http://nirgam.ecs.soton.ac.uk/>, 2007.
- [NMA⁺05] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet. Centralized Run-Time Resource Management in A Network-on-Chip Containing Reconfigurable Hardware Tiles. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 234 – 239 Vol. 1, 7-11 2005.
- [NMV⁺04] Vincent Nollet, Théodore Marescaux, Diederik Verkest, Jean-Yves Mignolet, and Serge Vernalde. Operating-System Controlled Network on Chip. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 256–259, New York, NY, USA, 2004.
- [Nox08] Noxim NoC Simulator, <http://sourceforge.net/projects/noxim>, 2008.
- [NPJS10] Mahesh Nanjundappa, Hiren D. Patel, Bijoy A. Jose, and Sandeep K. Shukla. SCGPSim: A Fast SystemC Simulator on GPUs. In *ASP-DAC '10: Proceedings of Asia and South Pacific Design Automation Conference*, Taipei, Taiwan, China, pages 149 - 154, 2010.

REFERENCES

- [NS01] S. Nussbaum and J.E. Smith. Modeling Superscalar Processors via Statistical Simulation. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 15 –24, 2001.
- [OCF00] M. Oskin, F.T. Chong, and M. Farrens. HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 71 – 82, 2000.
- [OH05] Kunle Olukotun and Lance Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, 2005.
- [OM06] Umit Y. Ogras and Radu Marculescu. Prediction-Based Flow Control for Network-on-Chip Traffic. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 839–844, New York, NY, USA, 2006.
- [OM08] Umit Y. Ogras and Radu Marculescu. Analysis and Optimization of Prediction-Based Flow Control in Networks-on-Chip. *ACM Trans. Des. Autom. Electron. Syst.*, 13(1):1–28, 2008.
- [OMCM07] Umit Y. Ogras, Radu Marculescu, Puru Choudhary, and Diana Marculescu. Voltage-Frequency Island Partitioning for GALS-Based Networks-on-Chip. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 110–115, New York, NY, USA, 2007.
- [OMP⁺05] Luciano Ost, Aline Mello, José Palma, Fernando Moraes, and Ney Calazans. MAIA: A Framework for Networks on Chip Generation and Verification. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 49–52, New York, NY, USA, 2005.
- [ONH⁺96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for A Single-Chip Multiprocessor. In *7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1996.
- [Ope07] OpenSPARC. OpenSPARC Portal, <http://www.opensparc.net/>, 2007.

REFERENCES

- [Ope10] Opencores.org. Wishbone System-on-Chip (SoC) Interconnect Architecture, <http://www.opencores.org/opencores,wishbone>, 2010.
- [OSC09] OSCI. The Open SystemC Initiative (OSCI), OSCI Standards and Reference Implementation, <http://www.systemc.org>, 2009.
- [Ote04] P. Otellini. Intel Keynote. In *Intel Developer Forum*, 2004.
- [PCC⁺09] Ezudheen P, Priya Chandran, Joy Chandra, Biju Puthur Simon, and Deepak Ravi. Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines. In *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 80–87, Washington, DC, USA, 2009.
- [PD01] Li-Shiuan Peh and William J. Dally. A Delay Model and Speculative Architecture for Pipelined Routers. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 255, Washington, DC, USA, 2001.
- [PdWSvM06] Milan Pastrnak, Peter H. N. de With, Sander Stuijk, and Jef van Meerbergen. Parallel Implementation of Arbitrary-Shaped MPEG-4 Decoder for Multiprocessor Systems. *Proc. Visual Comm. and Image Processing*, 2006.
- [PFH⁺06] D.A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D.I. August, and D. Connors. Exploiting Parallelism and Structure to Accelerate The Simulation of Chip Multi-Processors. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 29–40, Feb. 2006.
- [PGJ⁺05] Partha Pratim Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures. *Computers, IEEE Transactions on*, 54(8):1025–1040, Aug. 2005.
- [PI09] PI. An MPI Program Calculating The Value of π , <http://www-unix.mcs.anl.gov/mmpi/usingmpi/examples/simplempi/cpi.c.htm>, 2009.

REFERENCES

- [PJ09] Li-Shiuan Peh and Natalie Enright Jerger. *On-Chip Networks*. Morgan and Claypool Publishers, 2009.
- [PMT04] Daniel Gracia Pérez, Gilles Mouchard, and Olivier Temam. A New Optimized Implementation of The Systemc Engine Using Acyclic Scheduling. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, pages 552 - 557, Washington, DC, USA, 2004.
- [Pow04] Gomaa, Mohamed and Powell, Michael D. and Vijaykumar, T. N.. Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through The Operating System. In *SIGPLAN Notices*, 39(11), pages 260–270, 2004.
- [PS05] H.D. Patel and S.K. Shukla. Towards A Heterogeneous Simulation Kernel for System-Level Models: A SystemC Kernel for Synchronous Data Flow Models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(8):1261–1271, Aug. 2005.
- [RDP⁺05] A. Radulescu, J. Dielissen, S.G. Pestana, O.P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An Efficient On-Chip NI Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Configuration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):4–17, Jan. 2005.
- [RFT⁺05] Jose Renau, Basilio Fraguera, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC Simulator, January 2005. <http://sesc.sourceforge.net>.
- [SA05] Lawrence Spracklen and Santosh Abraham. Chip Multithreading: Opportunities and Challenges. In *HPCA-11*, 2005.
- [SBV95] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, 1995.
- [SC06] M. Saldana and P. Chow. TMD-MPI: An MPI Implementation for Multiple Processors across Multiple FPGAs. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, Aug. 2006.

REFERENCES

- [SD06] Li Shang and R.P. Dick. Thermal Crisis: Challenges and Potential Solutions. *Potentials, IEEE*, 25(5):31–35, Sept.-Oct. 2006.
- [SHW96] Bill Hay Steve Hoxey, Faraydon Karim and Hank Warren. *The PowerPC Compiler Writer’s Guide*. IBM, 1996.
- [Ska03] K. Skadron. Temperature-Aware Microarchitecture. In *Proceedings 30th ISCA*, 2003.
- [SMA⁺03] K. Skadron, M. Martonosi, D.I. August, M.D. Hill, D.J. Lilja, and V.S. Pai. Challenges in Computer Architecture Evaluation. *Computer*, 36(8):30–36, Aug. 2003.
- [SNKB01] Karthikeyan Sankaralingam, Ramadass Nagarajan, Stephen W. Keckler, and Doug Burger. SimpleScalar Simulation of The PowerPC Instruction Set Architecture. In *UT Austin Technical Report: CS-TR-00-04*, 2001.
- [SOC04] Timothy Sherwood, Mark Oskin, and Brad Calder. Balancing Design Options with SHERPA. In *CASES ’04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 57–68, New York, NY, USA, 2004.
- [SP96] Debendra Das Sharma and Dhiraj K. Pradhan. Submesh Allocation in Mesh Multicomputers Using Busy-List: A Bestfit Approach with Complete Recognition Capability. *Journal of Parallel and Distributed Computing*, 36(2), pages 106–118, 1996.
- [SSG02] N. Savoiu, S. Shukla, and R. Gupta. Automated Concurrency Re-Assignment in High Level System Models for Efficient System-Level Simulation. In *DATE ’02: Proceedings of the conference on Design, automation and test in Europe*, page 875, Washington, DC, USA, 2002.
- [SSS⁺04] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-Aware Microarchitecture: Modeling and Implementation. In *ACM Trans. Archit. Code Optim.*, volume 1, pages 94–125, New York, NY, USA, 2004.

REFERENCES

- [Sys10] Azul Systems. Vega 3 Series Processors, <http://www.azulsystems.com/products>, 2010.
- [TCC⁺00] Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. The MAJC Architecture: A Synthesis of Parallelism and Scalability. In *IEEE Micro*, 2000.
- [TCR⁺09] Sergio V. Tota, Mario R. Casu, Massimo Ruoch, Luca Macchiarulo, and Maurizio Zamboni. A Case Study for NoC-Based Homogeneous MPSoC Architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(3):384–388, 2009.
- [TEL95] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. 22nd Ann. Int’l Symp. Computer Architecture*, 1995.
- [Til09] Tiler. Tiler, <http://www.tiler.com>, 2009.
- [Tra04] M. Trams. A First Mature Revision of A Synchronization Library for Distributed RTL Simulation in SystemC, <http://www.digital-force.net>, 2004.
- [UNI09] UNISIM. United simulation environment, <https://unisim.org>, 2009.
- [VHR⁺07] Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, Shailendra Jain, Sriram Venkataraman, Yatin Hoskote, and Nitin Borkar. An 80-Tile 1.28Tflops Network-on-Chip in 65nm CMOS. 2007.
- [VSG⁺10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing The Energy of Mature Computations. In *ASPLOS ’10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 205–218, New York, NY, USA, 2010.
- [VVP⁺02] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, and David I. August. Microarchitectural Exploration with Liberty. In *MICRO*

- 35: *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 271–282, Los Alamitos, CA, USA, 2002.
- [VVP⁺06] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, Sharad Malik, and David I. August. The Liberty Simulation Environment: A Deliberate Approach to High-Level System Modeling. *ACM Trans. Comput. Syst.*, 24(3):211–249, 2006.
- [Wal91] David W. Wall. Limits of Instruction-Level Parallelism. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, New York, NY, USA, 1991.
- [WB06] A. Weissel and F. Bellosa. Balancing Power Consumption in Multiprocessor Systems. In *Proceedings of the First ACM SIGOPS EuroSys*, 2006.
- [WGH⁺07] 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. *Micro, IEEE*, 27(5):15–31, Sept.-Oct. 2007.
- [Wis07] Wisconsin. GEMS Simulator, 2007. <http://www.cs.wisc.edu/gems/>.
- [WJY⁺07] Wei Wu, Lingling Jin, Jun Yang, Pu Liu, and Sheldon X.-D. Tan. Efficient Power Modeling and Software Thermal Sensing for Runtime Temperature Monitoring. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–29, 2007.
- [WM95] Wm. A. Wulf and Sally A. McKee. Hitting The Memory Wall: Implications of The Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.
- [Wol04] Wayne Wolf. The Future of Multiprocessor Systems-on-Chips. In *DAC 2004*, 2004.
- [WSJZ09] Ling Wang, Hui Song, Yingtao Jiang, and Lihong Zhang. A Routing-Table-Based Adaptive and Minimal Routing Scheme on Network-on-Chip Architectures. *Comput. Electr. Eng.*, 35(6):846–855, 2009.

REFERENCES

- [WZPM02] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *Proceedings 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, 2002.
- [YEL⁺06] J.J. Yi, L. Eeckhout, D.J. Lilja, B. Calder, L.K. John, and J.E. Smith. The Future of Simulation: A Field of Dreams. *Computer*, 39(11):22–29, Nov. 2006.
- [YMB02] Terry Tao Ye, Giovanni De Micheli, and Luca Benini. Analysis of Power Consumption on Switch Fabrics in Network Routers. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, pages 524–529, New York, NY, USA, 2002.
- [Zhu92] Y. Zhu. Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. *Journal of Parallel and Distributed Computing*, 16(4), pages 328–337, 1992.
- [ZL01] Jipeng Zhou and F.C.M. Lau. Adaptive Fault-Tolerant Wormhole Routing in 2D Meshes. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 8 pp.–, Apr 2001.
- [ZLJ05] Mikael Millberg Erland Nilsson Zhonghai Lu, Rikard Thid and Axel Jantsch. NNSE: Nostrum Network-on-Chip Simulation Environment. in DATE 2005 university booth tool demonstration, Munich, Germany, <http://www.imit.kth.se/info/fofu/nostrum/>, 2005.
- [ZS09] Yufu Zhang and Ankur Srivastava. Accurate Temperature Estimation Using Noisy Thermal Sensors. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 472–477, New York, NY, USA, 2009.