TURUN YLIOPISTO
UNIVERSITY OF TURKU

RUN-TIME MANAGEMENT OF MANY-CORE SOCS

A communication-centric approach

Mohammad Fattah

TURUN YLIOPISTON JULKAISUJA – ANNALES UNIVERSITATIS TURKUENSIS
SARJA – SER. F OSA – TOM. 7 | TECHNICA – INFORMATICA | TURKU 2021

TURUN
YLIOPISTO
UNIVERSITY
OF TURKU

RUN-TIME MANAGEMENT OF MANY-CORE SOCS

A communication-centric approach

Mohammad Fattah

TURUN YLIOPISTON JULKAISUJA – ANNALES UNIVERSITATIS TURKUENSIS
SARJA – SER. F OSA – TOM. 7 | TECHNICA – INFORMATICA | TURKU 2021

University of Turku

Faculty of Technology
Department of Computing
Information and Communication Technology
Doctoral Programme in Mathematics and Computer Sciences (MATTI)

Supervised by

Professor Juha Plosila
University of Turku

Professor Pasi Liljeberg
University of Turku

Professor Hannu Tenhunen
University of Turku

Reviewed by

Professor Jüri Vain
Tallinn University of Technology

Professor Timo D. Hämäläinen
Tampere University

Opponent

Professor Peeter Ellervee
Tallinn University of Technology

ISBN 978-951-29-8708-5 (PRINT)
ISBN 978-951-29-8709-2 (PDF)
ISSN 2736-9390 (PRINT)
ISSN 2736-9684 (ONLINE)
Painosalama, Turku, Finland, 2021

Mom...

UNIVERSITY OF TURKU
Faculty of Technology
Department of Computing
Information and Communication Technology
FATTAH, MOHAMMAD: Run-Time Management of Many-Core SoCs
Doctoral dissertation, 151 pp.
Doctoral Programme in Mathematics and Computer Sciences (MATTI)
November 2021

ABSTRACT

The single core performance hit the power and complexity limits in the beginning of this century, moving the industry towards the design of multi- and many-core system-on-chips (SoCs). The on-chip communication between the cores plays a critical role in the performance of these SoCs, with power dissipation, communication latency, scalability to many cores, and reliability against the transistor failures as the main design challenges. Accordingly, we dedicate this thesis to the communication-centered management of the many-core SoCs, with the goal to advance the state-of-the-art in addressing these challenges. To this end, we contribute to on-chip communication of many-core SoCs in three main directions.

First, we start with a synthesizable SoC with full system simulation. We demonstrate the importance of the networking overhead in a practical system, and propose our sophisticated network interface (NI) that offloads the work from SW to HW. Our results show around 5x and up to 50x higher network performance, compared to previous works. As the second direction of this thesis, we study the significance of run-time application mapping. We demonstrate that contiguous application mapping not only improves the network latency (by 23%) and power dissipation (by 50%), but also improves the system throughput (by 3%) and quality-of-service (QoS) of soft real-time applications (up to 100x less deadline misses). Also our hierarchical run-time application mapping provides 99.41% successful mapping when up to 8 links are broken. As the final direction of the thesis, we propose a fault-tolerant routing algorithm, the maze-routing. It is the first-in-class algorithm that provides guaranteed delivery, a fully-distributed solution, low area overhead (by 16x), and instantaneous reconfiguration (vs. 40K cycles down time of previous works), all at the same time. Besides the individual goals of each contribution, when applicable, we ensure that our solutions scale to extreme network sizes like 12x12 and 16x16. This thesis concludes that the communication overhead and its optimization play a significant role in the performance of many-core SoCs.

KEYWORDS: Network-on-chip, Many-core SoCs, Run-time application mapping, Fault-tolerance, Routing algorithm

TURUN YLIOPISTO
Teknillinen tiedekunta
Tietotekniikan laitos
Tieto- ja viestintätekniikka
FATTAH, MOHAMMAD: Run-Time Management of Many-Core SoCs
Väitöskirja, 151 s.
Matemaattis-tietotekninen tohtoriohjelma (MATTI)
Marraskuu 2021

TIIVISTELMÄ

Yksittäisen ytimen suorituskyky on rajoittunut tehonkulutuksen ja kompleksisuuden osalta. Ytimien välinen viestinvälitys vaikuttaa merkittävästi järjestelmäpiirin suosituskykyyn, tehonkulutukseen, viipeeseen, skaalautuvuuteen ja luotettavuuteen. Toimiala on siirtynyt käyttämään moniydin- (engl. multi-core) ja tosimoniydinratkaisuihin (engl. many-core) perustuvia järjestelmäpiirejä (engl. System-on-Chip). Tämä väitöskirja keskittyy tosimoniydinjärjestelmien viestinvälityksen haasteisiin ja käsittelee kolmea keskeistä tosimoniydinjärjestelmän viestinvälityksen haastetta.

Ensimmäiseksi tutkimme syntetisoituvia järjestelmäpiirejä järjestelmäsimuloinneilla. Osoitamme, että toimivassa järjestelmässä viestienvälityksen kustannus (engl. overhead) on merkittävä ja esittelemme kehittämämme verkkorajapinnan ohjelmistojen ja verkon välille, jossa työtä siirretään laitteistolta ohjelmistojen suoritettavaksi. Tulokset osoittavat noin viisinkertaisen ja jopa viisikymmenkertaisen suorituskykyparannuksen verrattuna aikaisempiin töihin. Toiseksi tutkimme ajonaikaista sovelluksen mappauksen (engl. mapping) merkitystä tosimoniydinjärjestelmäpiirin sisäisessä viestinvälityksessä. Osoitamme, että jatkuva sovelluksen mappaus ei pelkästään paranna verkon latenssia (23 %) ja tehonkulutusta (50 %), vaan myös parantaa lievennetyssä reaaliaikaisuudessa (engl. soft real-time) järjestelmän läpäisyä (3 %) ja palvelunlaatua (jopa 100 kertaa vähemmän määräajan ylityksiä). Osoitamme myös, että esittelemänne hierarkinen ajonaikainen sovellusmappaus tuottaa 99,41 % onnistumisen, kun jopa kahdeksan viestilinkkiä on pois käytöstä. Viimeiseksi ehdotamme vikasietoista reititysalgoritmia, joka tarjoaa samanaikaisesti useita hyödyllisiä ominaisuuksia: taattu toimitus, täysin hajautettu toteutus, pieni pinta-ala ja uudelleenkonfiguroinnin matala kustannus. Edellä mainittujen löydösten lisäksi skaalautuvuus on tärkeää ottaa huomioon. Väitöskirjassa esitetyt ratkaisut skaalautuvat aina 12x12 ja 16x16 viestintäverkkojen kokoon asti. Väitöskirja toteaa viestinvälityksen kustannuksen ja viestinvälityksen optimoinnin vaikuttavan merkittävästi tosimoniydinjärjestelmäpiirien suorituskykyyn.

ASIASANAT: tosimoniydinjärjestelmäpiiri, ajonaikainen sovelluksen mappaus, vikasietoisuus, reititysalgoritmi

Acknowledgements

I carried out the research in this dissertation at the Department of Computing, University of Turku, from March 2011 to December 2015. However, it took me another six years to finalize my dissertation as several pivotal events occurred in my personal life during that period. Looking back, an abundance of people and organizations supported me both professionally and personally, and I am sincerely grateful to them.

First of all, I would like to express my gratitude to my supervisors Professor Juha Plosila and Professor Pasi Liljeberg, and my research director Professor Hannu Tenhunen. I started as a young researcher, and throughout the years, their inspiration, wisdom and encouragement has helped me to become who I am today. Juha and Pasi, it was priceless to not only have your guidance during my research, but also to have your support throughout the tough and rough times.

I would like to thank Professor Jüri Vain from Tallinn University of Technology and Professor Timo D. Hämäläinen from Tampere University of Technology, for their detailed reviews and constructive comments on my dissertation.

The Graduate School in Electronics, Telecommunication and Automation (GETA) is gratefully acknowledged. In addition to funding my doctoral studies, it supported the two valuable and unforgettable research visits I had during this time. I am also grateful to the Ulla Tuominen Foundation and Elisa HPY Research Foundation for financially supporting my research.

I had the privilege to conduct two research visits during this time, which led to two of the publications presented int this thesis. I wish to express my great appreciation to Associate Professor Maurizio Palesi from University of Catania for his guidance and supervision. I would like to thank him, Salvatore, Paolo, Antonio, Davide and others for making me feel at home. Also, I wish to express my great appreciation to Professor Onur Mutlu from Carnegie Mellon University for his guidance and valuable time. I would like to thank him, Samira, Yoongu, Justin, Yixin, Lavanya, Vivek, and others for the many interesting discussions we had, and all that I learned from them.

I would like to thank all of my co-authors for their efforts, insights, and fruitful collaboration. In particular, I am grateful to Masoud Daneshtalab for his valuable contributions, Hashem Haghbayan for our valuable discussions, Antti Airola and Tapio Pahikkala for their great algorithmic insights, and Rachata Ausavarungnirun for being a great collaborator. I would like to thank Ali Hazmi and Kari Pietikäinen, my colleagues at Nokia, for giving me one last push to bring this thesis to completion. I would like to thank my dear friends, Maria and Jarkko for all the fun we had together, Johanna and Matti for being fantastic friends, and my Iranian friends, Maryam, Faezeh, Sanaz, Mojtaba, Hossein, Ali, Erfan, and Farrokh for the great times we spent together.

It is impossible to imagine my dissertation without the love, patience and support of my parents. I am blessed to have parents that valued education, and sacrificed so much to support me. Their support and encouragement over the years has been the greatest gift I have ever had. My great dad (extended to Abbasali), I am so lucky to have your support and delicate care to shield me against all distractions and adversities. My perfect mom (extended to Parvaneh– butterfly in English), you always offered me the peace of mind and confidence in all that was out there. I wish I could share this very moment with you. This thesis' dedication extends to your memory, who would be with me forever. Zahra, Zohreh, and Ali, my beloved siblings, thank you for your encouragements and most importantly for being at home, when I have been away.

Finally, and with all my heart, I am grateful to my family in Finland. Narges, my adorable love, you have been an amazing friend ever since we met in Turku, an understanding and caring wife, and an incredible mother. Thank you for walking hand-in-hand with me through thick and thin. Your support and encouragement have been essential during the course of this dissertation. Soren, my happy little boy, you might be still too young to follow all the details. But when you read these words, I would like you to know that no matter what, I will always love you so much.

November 2021
Mohammad Fattah

Table of Contents

Abbreviations

AM	Application Manager	18, 45–49
AXI	Advanced Extensible Interface	24
BN	Best Neighbor	31
CASqA	Contiguity Adjustable Square Allocation	40–42, 47, 48, 50, 64
CoNA	Contiguous Neighborhood Allocation	20, 35, 36, 38, 39, 58, 59
CPU	Central Processing Unit	1, 2, 4, 16, 27, 30, 33, 34, 42
DMA	Direct Memory Access	27
DTL	Device Transaction Level	24
ENIAC	Electronic Numerical Integrator And Computer	1
FF	First Free	31
FIFO	First-in First-out	26
flit	Flow Control Digit	23
FLOPS	Floating Point Operations Per Second	2
FPGA	Field-programmable Gate Array	27, 28, 32
GALS	Globally-asynchronous Locally-synchronous	23
GEA	Gaussian Elimination Application	24, 25, 30
IC	Integrated Circuits	1
ILP	Instruction-level Parallelism	2–4
IP	Intellectual Property	24
IPC	Instruction Per Cycle	2
LBDR	Logic-based Distributed Routing	52

LBDRe LBDR-extended 52

LLC Last Level Cache 64

MD Manhattan Distance 34, 54

MM Mobile Master 18, 45–48

MPI Message Passing Interface 24

MPSoC Multi-processor System-on-chip 14, 20, 32

MRD Mapped Region Dispersion 34, 35

MTTF Mean-time-to-failure 13

NI Network Interface . . . . v, 14, 15, 20, 23, 24, 26–28, 58

NMRD Normalized MRD 35

NN Nearest Neighbor 31, 39, 41

NoC Network-on-chip . 12–21, 23, 24, 27, 30–35, 38, 39, 41, 42, 45, 47, 51, 52, 61, 63

OCP Open Core Protocol 24

OS Operating System 23, 25, 27, 40, 58, 60

PE Processing Element 12, 14, 16–18, 24, 26, 27, 30, 32–34, 39, 42, 45, 47, 48, 53, 63

PL Path Load 31

QoS Quality-of-service . . . . v, 10, 17, 42, 50, 63

RQ Research Question 12, 13, 21

RTL Register-transfer Level 27, 33

SCC Single-chip Cloud Computer 14, 28, 63

SF Square Factor 36, 38, 44

SHiC Smart Hill Climbing 35, 38–40, 46, 48, 59

SHiFA System-level Hierarchical Fault-aware 45, 47–50, 63

SoC System-on-chip . . . . v, 27, 63, 64

TG Task Graph 49

uLBDR Universal-LBDR 52

VIP Virtual Point-to-point 32

List of Original Publications

This dissertation is based on the following original publications, which are referred to in the text by their Roman numerals. Some ideas and figures of the dissertation might have appeared previously in the these publications.

I Mohammad Fattah, Masoud Daneshtalab, Pasi Liljeberg, Juha Plosila. Exploration of MPSoC Monitoring and Management Systems. In *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–3, 20–22 June 2011, Montpellier, France.

II Mohammad Fattah, Masoud Daneshtalab, Pasi Liljeberg, Juha Plosila. Transport Layer Aware Design of Network Interface in Many-Core Systems. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–7, 9–11 July 2012, York, UK.

III Mohamamd Fattah, Marco Ramirez, Masoud Daneshtalab, Pasi Liljeberg, Juha Plosila. CoNA: Dynamic Application Mapping for Congestion Reduction in Many-Core Systems. In *IEEE 30th International Conference on Computer Design (ICCD)*, pp. 364–370, Sept. 30 2012–Oct. 3 2012, Montreal, QC, Canada.

IV Mohammad Fattah, Masoud Daneshtalab, Pasi Liljeberg, Juha Plosila. Smart Hill Climbing for Agile Dynamic Mapping in Many-Core Systems. In *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, May 29 2013–June 7 2013, Austin, TX.

V Mohammad Fattah, Pasi Liljeberg, Juha Plosila, Hannu Tenhunen. Adjustable Contiguity of Run-Time Task Allocation in Networked Many-Core Systems. In *19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 349–354, 20–23 Jan. 2014, Singapore.

VI Mohammad Fattah, Amir-Mohammad Rahmani, Thomas Canhao Xu, Anil Kanduri, Pasi Liljeberg, Juha Plosila, Hannu Tenhunen. Mixed-Criticality Run-Time Task Mapping for NoC-Based Many-Core Systems. In *22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 458–465, 12–14 Feb. 2014, Torino, Italy.

VII Mohammad-Hashem Haghbayan, Amir-Mohamad Rahmani, Antonio Miele, Mohammad Fattah, Juha Plosila, Pasi Liljeberg, Hannu Tenhunen. A Power-Aware Approach for Online Test Scheduling in Many-core Architectures. In *IEEE Transactions on Computers*, no. 99, pp. 730–743.

VIII Mohammad Fattah, Maurizio Palesi, Pasi Liljeberg, Juha Plosila, Hannu Tenhunen. SHiFA: System-Level Hierarchy in Run-Time Fault-Aware Management of Many-Core Systems. In *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 1–5 June 2014, San Francisco, CA.

IX Mohammad Fattah, Antti Airola, Rachata Ausavarungnirun, Nima Mirzaei, Pasi Liljeberg, Juha Plosila, Siamak Mohammadi, Tapio Pahikkala, Onur Mutlu, Hannu Tenhunen. A Low-Overhead, Fully-Distributed, Guaranteed-Delivery Routing Algorithm for Faulty Network-on-Chips. In *9th International Symposium on Networks-on-Chip (NOCS)*, pp. 18:1–18:8, 28–30 Sept. 2015, Vancouver, BC, Canada.

1 Introduction

Since the early days of digital computers, mankind has always been seeking for more and more computing power. Advances in computer design has enabled the development of programs and applications that were hard, if not impossible, to imagine in early days of computers. Many scientific discoveries and advances in different aspects of human life (such as economy, health, etc.) has become real because of the increasing computing power that has become increasingly time and price affordable from day to day.

Early digital computers were *electromechanical*, where calculations were performed by driving mechanical relays using electric switches. In 1941, the German engineer Konard Zuse invented Z3, the world's first *electromechanical* programmable digital computer [99; 108]. 2000 relays were incorporating in Z3 to perform 22-bit arithmetic at clock frequency of 5 to 10 Hz. Nevertheless, *all-electric* computers quickly replaced the electromechanical computers, caused by the invention of vacuum tubes.

In 1946, electronic numerical integrator and computer (ENIAC) [31] was announced as the world's first electronic general purpose computer. 17,468 vacuum tubes along with several crystal diodes, relays, resistors and capacitors were building up ENIAC to perform 10 digit operations at clock frequency of 5000 Hz; thousand times faster than Z3– the electromechanical computer.

Later on, the invention of bipolar transistors (in 1947), and integrated circuitss (ICs)— in 1959, revolutionized the use and computing power of digital computers. The IC design enabled the integration of the central processing unit (CPU) of a digital computer into one IC (or at most few), called a *microprocessor*. In 1971, Intel announced the world's first microprocessor available in the market, the Intel 4004 [1]. 2,300 transistors of 10 μm feature size were put together to perform 4-bit operations at clock frequency of 740 kHz. This offered a computing power almost equal to that of ENIAC, with several orders of magnitude less weight, power consumption, and size characteristics[1].

Ever since, as predicted by Gordon E. Moore at 1965 [72] (known as Moore's law), the number of transistors integrated in a single chip has been doubled almost every 18 months. This is realized by continues shrinkage in the feature size of transis-

[1]ENIAC was 27,000 kg, consuming 150kW of electricity and occupying 167 m^2 of area

tors, as well as advances in manufacturing technologies. To keep up the performance of microprocessors with the number of integrated transistors, paradigm shifts and design innovations are being deployed to mitigate new challenges and bottlenecks introduced as a result of the transistor scaling. Continuous research is needed to predict, identify and find solutions to challenges and bottlenecks of delivering steady performance improvements with regards to increasing transistor count. This thesis is willing to be a very small step in this direction.

1.1 Background

Until the early days of this century, microprocessors used to have a single CPU, known as single-core processors or uniprocessors. Essentially, there was no need to put more CPUs into a microprocessor, because their speed, measured as floating point operations per second (FLOPS), used to follow the Moore's law by exploiting two main techniques; instruction-level parallelism (ILP), and clock frequency scaling [51]. ILP methods try to potentially overlap and parallelize execution of different instructions, leading to execution of more instruction per cycle (IPC). Example ILP methods are using deeper pipelines, larger out of order execution buffers, higher instruction-fetch bandwidth (i.e. superscalar architectures), and so on. The design complexity and transistors count increase with the degree of utilized ILP. That said, the continuation of Moore's law worked as an enabler to higher IPC values. The frequency scaling, on the other hand, is the other main reason for speedup of microprocessors. Capacitance of transistors decrease with their feature size, in which, they can switch in a higher frequency. As a result of IPC and frequency improvements, the speed of microprocessors increased with the number of integrated transistors; i.e. followed the Moore's law.

Although Moore's law was still in place, around 2005, the performance of single-core processors became difficult to improve. First, the gains of ILP methods greatly diminished. Not only there is a limited amount of ILP in a single instruction stream to harvest [51], but also, fundamental circuit limitations manifest when increasing the degree of utilized ILP methods [78]. Moreover, the frequency scaling, as the second knob for higher speed, has hit to the power wall [14]. That is, scaling the frequency in smaller technology nodes generates heat at a rate faster than can be transferred to the ambient. As technology scales further, the reduction of supply voltage (V_{DD}) is slowed down and leakage current portion is increased. Thus, in order for transistors to switch in a faster frequency,they should be supplied with higher voltage (V_{DD}) than expected. With respect to the V_{DD}value, however, the leakage (a.k.a. static: $P_{stat.}$) and switching (a.k.a dynamic: $P_{dyn.}$) power of transistors increase linearly and quadratically, respectively. Since increasing the frequency ($Freq.$) requires linear increase of V_{DD}, the operating frequency has linear and power of 3 relation, respectively, to the static and dynamic power consumption of a processor:

$$P_{stat.} \propto Freq. \\ P_{dyn.} \propto Freq.^3 \tag{1}$$

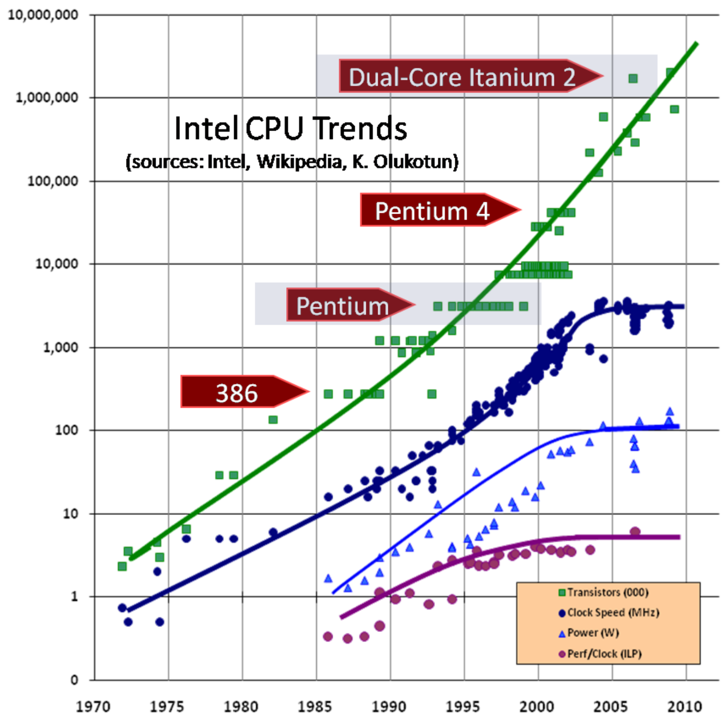


Figure 1. Frequency and ILP gains stopped, but Moore's law continued [94].

In summary, increasing the degree of ILP methods faced diminishing returns and increasing microprocessors frequency led to excessive heat dissipation, making the case for higher speed in single-core microprocessors impractical and expensive. Fig. 1 shows the above mentioned trend, for Intel microprocessors, since 1970 until 2010. As can be seen, the Moore's law has been in place and the total number of transistors has been increasing exponentially (green data points). However, around 2003, the ILP performance reached its limits, shown as Perf/Clock trend line, where no more performance is achieved per clock cycle using ILP methods. In addition, the frequency scaling of microprocessors has stopped (dark blue line) around the same time.

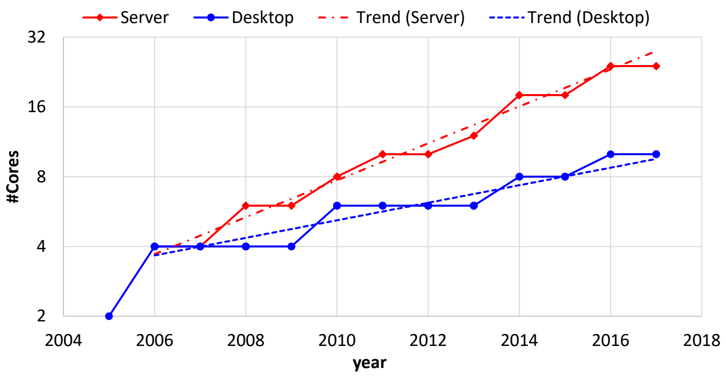


Figure 2. The maximum core count available by Intel in each year.[2]

1.1.1 Many-Core Architectures

As mentioned earlier, around 2005, the performance of single-core microprocessors started to lag behind the capacity offered by Moore's law. As a result, industry experienced a big paradigm shift towards the multi-core processor design. Instead of using the additional transistor capacity to design a complexer uniprocessor, it is used to accommodate multiple of simpler cores. Theoretically, this multiplies the performance of the microprocessor design, while ILP and frequency scaling limitations are removed from the equation.

Since then, the number of cores in a single processor is gradually increasing, to the point where we do not have only multiple but *many* cores on a single chip. To better illustrate this trend, we have gathered the information of different CPU models offered by Intel in every year. CPUs are partitioned into two main categories, the desktop processors and the server processors (called Xeon processors by Intel). Fig. 2 demonstrates the maximum number of cores available in a single processor by each year. As can be seen from the trend lines, the number of cores is double every 8 years in desktop market, while doubled every 4 years in server processors.

1.1.2 Workloads of Many-Core Architectures

In single-core era, software was written sequentially; as a sequence of machine instructions. It was the hardware responsibility to dynamically find instructions that are independent and parallelize (overlap) their execution. Parallel execution of instructions is allowed as long as it keeps the illusion of a sequential processor to the software side. Detecting the dependencies between different instructions and correctly scheduling their execution is a difficult task. ILP methods ease the software

[2] The data is extracted from CPU-World website: http://www.cpu-world.com/

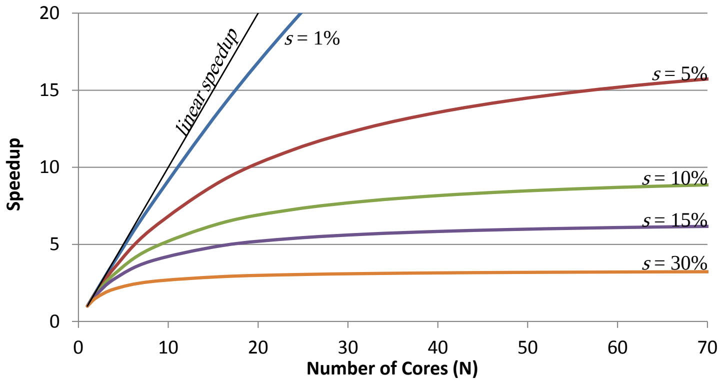


Figure 3. According to Amdahl's law, the achievable speed up is severely limited by serial bottleneck.

development, as programmers do not need to think in parallel. Software programs are developed without considering different dependencies and the hardware takes care of the rest in providing high-speed execution.

By end of single-core era and introduction of multi-core processors, however, the available parallelism is exposed to the programmers. Now, programmers need to think in parallel, partition an application into parallel tasks and take care of the dependencies between them. Accordingly, a parallelized application will be a set of smaller tasks that communicate and cooperate to solve the larger problem of the sequential application [3].

However, not all parts of an application are always parallelizable. Some parts, known as the serial portion (s), have to run sequentially; i.e. there are no other tasks to be executed in parallel with them. Accordingly, known as *Amdahl's law* or *serial bottleneck* [4], the maximum achievable speed up is limited by the serial portion of an application:

$$Speedup = \frac{1}{s + \frac{1-s}{N}} \tag{2}$$

Where, $Speedup$ is the maximum achievable speedup by running the application on N processors, compared to on 1 processor. The effect of the serial portion (s) is shown in Fig. 3. For instance, when there is only 5% of serial code, the maximum speedup is around 15 on a 64-core processor, where theoretically (i.e. $N \rightarrow +\infty$) it cannot exceed 20.

Hence, even though having many-(small-)cores rather than few-(large-)cores potentially provides a higher speedup, many-cores are not favorable as long as we want to parallelize *one* application. The high performance of many-core systems can be utilized by having several applications, where each is parallelized on few cores of the system [16]. To better visualize this, let us assume that for a given transistor count,

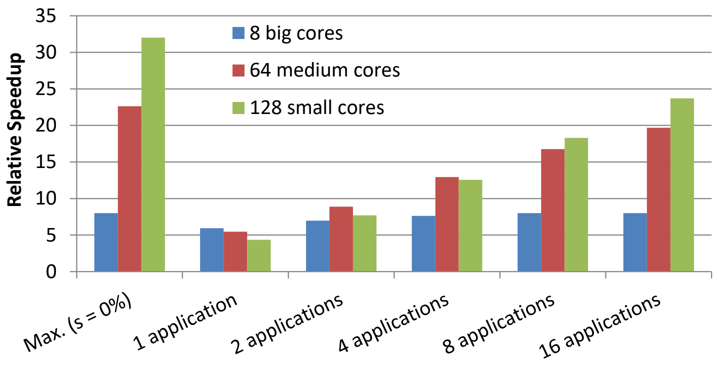


Figure 4. Relative speedup (compared to one big core) of running 1 to 16 applications ($s = 5\%$) on the same die size divided between different number of cores.

we have 3 microprocessor design options; to have 1) 8 big cores, 2) 64 medium cores, or 3) 128 small cores. Saying that one big core has a relative speed of 1, according to Pollack's rule, the relative speed of each medium and small core will be 0.35 and 0.25, respectively. As Amdahl's law implies, running an application with 5% serial portion on 8 powerful cores will be faster than running it on 128 cores which each is a fourth powerful. However, the story will be different if we have multiple applications. Fig. 4 shows the case where we have multiple applications ($s = 5\%$) running in parallel on each design option. As can be seen, 128 small cores demonstrate better speedup than 8 big cores as we increase the number of parallel applications. Note that the $Max.(s = 0\%)$ is the maximum speedup of each design option if we had 0% of serial portion.

In summary, one of the main distinguishing characteristics of many-core architectures is their workloads. Parallel applications, with different requirements and characteristics, enter and leave the system during run-time, with no a priori knowledge. One main example is the cloud computing servers where several users with different requirements (e.g. core count) share the same host machine; i.e. microprocessor.

Nevertheless, there are two main programming models for parallel tasks of an application to communicate, synchronize and cooperate; namely message-passing and shared-memory [3]. In message-passing, the communication happens through explicit messages, while a shared address space is used for communication in shared-memory model.

For instance, if task T_A wants to hand the result of its computation, X, to another task T_B, the following is done. In a shared-memory programming model, once T_A has the result ready in X sets a flag variable $flag$ so that the T_B can be notified and read the X value. As can be seen, in this scenario, we assume that X and $flag$ variables are shared between two parallel tasks. On the other hand, in a message-

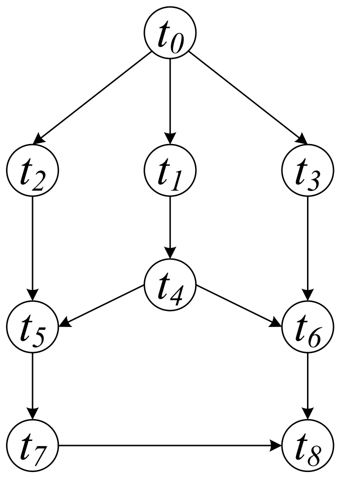


Figure 5. Gaussian Elimination application with 9 tasks and 11 edges

passing model, T_A will have an explicit $send$ command, which takes destination T_B and the data to be sent X as arguments. The receiver, T_B, also will use an explicit $receive$ command with the source T_A and the place to store the received data X_{copy} as arguments. As can be seen, in shared-memory model, the communication and synchronization is implicit and through shared memory space (variables), whereas in message-passing model, explicit communication primitives are used to exchange data while variables are private to each task. Note that parallel tasks of shared-memory applications are usually referred to as *threads*. We will use the same notion throughout this thesis; tasks are referring to parallel entities of a message-passing application, while threads are those of a shared-memory application.

Compared to message-passing model, where, programmer needs to take care of sharing required data among parallel tasks, the shared-memory model is closer to sequential programming, hence an easier choice for developers. On the other hand, and from the architecture design point of view, the networking overhead is optimized in message-passing, as only the required data is shared among different processors.

A message-passing application is often represented as a directed task graph $A_p = TG(T, E)$. A vertex $t_i \in T$ represents a task in the application, while edges $e_{i,j} \in E$ stand for communication from t_i to t_j. Task graph of Gaussian Elimination application [5] with 9 parallel tasks is shown in Fig. 5. Each edge $(e_{i,j})$ may also have a weight $w_{i,j}$ which determines communication volume between the two task. Task graphs can be built based on the application flow or by profiling it.

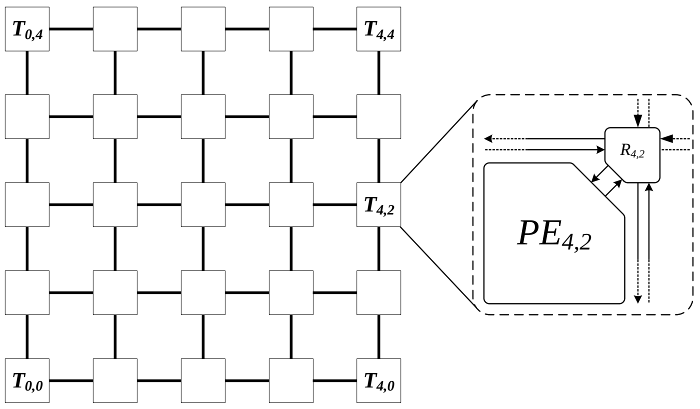


Figure 6. A 5x5 many-core architecture connected by a mesh network-on-chip.

1.1.3 Communication-Centric Design of Many-Core Architectures

As stated by Almasi [3], a parallel computer (a many-core architecture in our case) is a set of processing elements that can *communicate* and cooperate. Traditionally, shared buses are used as the communication infrastructure of microprocessors. As the bus is a shared medium, the throughput per core decreases with the increase of connected cores. Moreover, the wire length increases with the number of connected cores, which decreases the aggregated throughput. As a result, however, shared buses cannot scale to connect growing number of cores in many-core processors.

Network-on-chips (NoCs) have emerged as a promising solution for communication infrastructure of such systems [26]. Compared to traditional or hierarchical bus architectures, NoCs provide a scalable and regular infrastructure to connect components of many-core architectures. Instead of shared channels of bus architectures, NoCs are composed of routers and links. Each core is connected to a router, and injects messages to and receives them from the network through its router. Routers are connected together via links and forward messages from source core to the destination core. Fig. 6 shows a generalized 25-core architecture connected by a mesh on-chip network. Each core, encapsulated in a processing element (PE), can communicate with other cores through the network. In addition to the core, a PE may contain cache slices, a *network interface*, and so on.

NoC design is compromised of several aspects [27, Chapter 1]; namely, topology, routing algorithm, flow control and output scheduling. The network topology concerns the connection pattern between routers of the network, while the flow control mechanism addresses the way messages traverse a *path* in the network. The path taken by a message from source to destination is determined by the routing algorithm. Whilst, the output scheduling arbitrates between several messages that want to leave a router through the same link.

Similar to programming models of parallel applications, PEs of a many-core system may communicate in either message-passing or shared-memory models. Messages that are mainly exchanged between cores will vary depending on the utilized communication model. Coherency and synchronization messages dominate in the shared-memory model, while explicit messages of running tasks are exchanged in a message-passing NoC architecture.

Pros and Cons of Each Communication Model The message-passing and shared-memory programming and architecture models come with their particular benefits and costs. From programming point of view, shared-memory model is easier to utilize. As the memory address space is shared between threads, the programmer does not need to concern about partitioning of the data. Whereas in message-passing paradigm, the programmer needs to explicitly take care of data partitioning. Coherence protocols of shared-memory architectures, on the other hand, generate excessive network traffic for broadcasting messages, and require long tags to store sharers of each cache line. Both of which, impose scalability issues [69] when increasing the number of cores.

1.2 Design Challenges

As mentioned in Chapter 1.1.1, many-core architectures are proposed to mainly overcome the power issues of ever sophisticating single-core microprocessors. However, designing efficient many-cores leads to challenges of its own. These challenges may arise as a result of the massive parallelism of these architectures or aggressive shrinkage of the transistors feature size. This section explains these challenges, particularly from the on-chip communication point of view, which motivate the research questions of this thesis and its contributions.

1.2.1 Power

As mentioned, the power dissipation has been the main motivation to move from single-core microprocessors to multi- and many-core architectures. Compared to multi-cores, more number of smaller cores are integrated in a many-core system. As the size and complexity of a single core decreases, the relative size of the communication fabric (NoC) increases, resulting in increased share of NoC in the total power consumption of the chip. In 16-core Raw processor [97], for instance, 36% of the total chip power is dissipated in NoC fabric.

Hence, the NoC power consumption is a major challenge in design of many-core systems. Assuming a minimally-routed mesh network, as shown in [104; 103], the dynamic energy consumed to send a packet from a source node (src) to a destination

(dst) is:

$$E_{packet} = \#Flits \times E_{flit} \tag{3}$$

where, $\#Flits$ is the number of flits in that packet, while the energy consumed per flit of communication, indicated as E_{flit}, is:

$$\begin{aligned} E_{flit} &= E_R \times (MD(src, dst) + 1) + E_{wire} \times MD(src, dst) \\ &= MD(src, dst) \times (E_R + E_{wire}) + E_R \end{aligned} \tag{4}$$

where, E_R and E_{wire} are, respectively, the average energy consumed in each router and link of the path between source and destination. Also, $MD(src, dst)$ indicates the Manhattan Distance between source and destination; i.e. the path length between src and dst. As shown, the path length between the source and destination directly contributes to the dynamic power dissipation of the network.

Accordingly, the first motivation behind this thesis is to reduce the effect of power challenge in the on-chip network of many-core systems. We introduce several run-time application mapping algorithms that decrease the average hop counts that the network flits traverse in the network. Resulting in reduced power consumption of the NoC fabric.

1.2.2 Communication Latency

As explained in Chapter 1.1.2, Amdahl's law (equation 2) states that the speedup of a parallel application is limited by its serial portion; also know as serial bottleneck. What is not entered into the equation is the overhead of parallelizing the parallel portion. Amdahl's law assumes that the parallel portion of an application is perfectly parallelizable; i.e. it can be divided into N identical parts with no extra overhead.

Nevertheless, there are three main bottlenecks in parallelizing the parallel potion of an application. First, load imbalance: the parallel tasks may not have equal execution times. Resource contention in accessing the shared resources is the second bottleneck. Though we have N number of cores to run parallel tasks on them, not all the hardware resources are (can be) replicated.

The communication and synchronization overhead is the third bottleneck, which is the concern of this thesis. Parallel tasks of an application need to share and exchange data, which happens through the underlying on-chip network. The communication does not happen in zero-time, and the latency is increased by network contention. This adds up to the execution time of a parallel application and reduces the practical speedup. Moreover, the unpredictable communication latency can cause deadline misses in real-time applications, which reduces the QoS experienced by users. A real-time application is preferred to have predictable latencies with decreased worst-case latencies.

Accordingly, one of the motivations behind this thesis is to tackle this challenge. We aim to propose solutions that decrease the average and worst-case latency of the on-chip network. As explained, this potentially leads to improved execution time and QoS of the running applications and the many-core system.

Worth noting, however, as minimizing the serial portion of an application is more of a programming challenge, it falls beyond the scope of this thesis. Also, we assume that the serial bottleneck is backed up by running multiple parallel applications on a many-core system.

1.2.3 Reliability

Another major challenge of aggressive scaling of feature size is the reduced reliability of on-chip components [15]. The source of faults in the components can be either static or dynamic. Static faults happen during fabrication process and mainly occur due to device variations; i.e. differences in devices, from each other and what they are designed to be. Dynamic faults occur during run-time and can be permanent, transient, or intermittent [64]. Intermittent and transient faults mainly occur due to dynamic sources of variation as discussed in [15]. In intermittent faults, transistors timing and power characteristics fluctuates over time due to different dynamic mechanisms; e.g. change of the power density (Watts per square centimeter) across the chip area. This can, in turn, cause different components of the chip to periodically go faulty and back. Memory cells and logic latches are usually (and randomly) flipped in transient faults, caused by the alpha particles and cosmic rays that hit the silicon chips.

Permanent faults, on the other hand, occur due to different device aging and wear-out mechanisms [93; 57]; such as hot carrier injection (HCI), negative bias temperature instability (NBTI), electromigration, gate oxide breakdown, etc. These mechanisms are expected to be aggravated by technology shrinkage and cause more failed components during system operation, as devices age. In this thesis, we mainly focus on run-time permanent failures.

Hence, this is crucial to design different methods that test the system, detect faults, and adapt to them during run-time. The ultimate goals is to deliver reliable many-core systems with long lifetime, built from unreliable transistors and devices that fail much faster than the expected mean-time-to-failure (MTTF) of the whole system. The first step towards this goal is to detect such component failures early in time, by realizing some online test mechanisms. Online testing is required to ensure that the system is yet reliable and detect if a component has failed. When a failure is detected, run-time fault-tolerant methods are needed to adapt to the new situation and resume normal operation of system with minimal performance degradations.

Permanent faults may occur in PE or NoC component of a many-core system, wherein NoC faults can be more serious and challenging. Though faulty PEs degrade the system performance, they can be simply ignored as many-core systems provide inherent redundancy of PEs. However, a fault in a router or link of the on-chip network can potentially cripple system performance and perhaps even more severely become a single point of failure [45]; as NoC provides the communication substrate between the system components.

1.2.4 Scalability

Besides the above mentioned challenges, many-cores are recognized by their dynamic nature. Their workload, power dissipation, and utilization characteristics are dynamic and change during run-time, permanent faults occur randomly over time and so on. Hence, addressing aforementioned challenges cannot be complete without considering their dynamic nature.

Last but not least, we take scalability as one of the aims behind the proposals of this thesis. The proposed solutions need to scale by the core count, and increased dynamicity of the system. In other words, our power, speedup, and reliability performance needs to scale by the problem size.

1.3 Research Questions

The main motivation of this thesis is to address different challenges that the emergence of many-core architectures will raise (discussed in Chapter 1.2). Power issues are still going to be there, while speedup of parallel applications are not as promising as we wish, and it is hard to scale the monitoring and management algorithms to the many number of cores. In addition, reliability challenges get more severe as the technology is shrunk more and more. Scalability challenges raise in addressing these issues, especially when considering the dynamic nature of many-core systems. The communication substrate (realized by network-on-chips (NoCs)), as the main facilitator of the existing parallelism, has a major contribution in these challenges. NoC is going to have more contribution in the power dissipation of the whole system; as processing elements (PEs) get simpler and smaller. Execution time of running applications is affected by communication of their parallel tasks. While, failures in the NoC components can lead to a single point of failure. Accordingly, we set the objective of this thesis as follows:

> To investigate the main contributors in the *power*, *latency*, and *reliability* challenges of on-chip communication in many-core architectures, given their dynamic natures, and to propose *scalable* solutions to improve the state-of-the-art.

We set one or two research questions (RQs) in accordance to each design challenge to help this thesis in achieving its objective. Each RQ tries to narrow down the research focus, and pose questions on identifying the run-time bottlenecks or resolving the challenges.

Power

As mentioned in Chapter 1.2.1, the dynamic power dissipation of NoC fabric depends on the paths that packets traverse. The longer the path a packet takes, more power the NoC consumes. In the first RQ, we question the potential NoC power saving can be obtained through run-time placement of tasks on system cores, and try to come up with solutions to harvest this potential:

> RQ.1. How can we reduce the on-chip communication power through run-time application mapping algorithms?

Communication Latency

Execution time of a parallel application is affected by its communication latency, discussed in Chapter 1.2.2. There are several steps involved in communication between two tasks, contributing to the latency. The residing cores of tasks and their potential other sharer tasks are chosen dynamically during run-time. In this thesis, we question the main run-time dependent contributors to the network latency, and look for different architecture- and system-level solutions to reduce their share:

> RQ.2.1. What are the main run-time contributors to communication latency?

> RQ.2.2. How can we reduce impact of different contributors to communication latency, and how the system performance is affected?

Reliability

Aggressive scaling of transistor feature size has led to serious reliability challenges in the design of future many-core systems. System components may fail early in time, well before the expected mean-time-to-failure (MTTF) of the whole device. Online testing and run-time fault adaptation mechanisms are two main ingredients of designing fault-tolerant systems. Online testing methods periodically disable different system components to validate their correct functionality, which can potentially lead to degraded performance of the system. This thesis questions ways to reduce the degraded performance of online testing methods, and architecture- and system-level solutions to tolerate run-time permanent faults in NoC fabric components:

> RQ.3.1. How to incorporate online testing methods providing minimal run-time performance degradation?

RQ.3.2. How can we recover from failures in the NoC components?

Scalability

Many-core systems will be dynamic and variant in nature in many different aspects. Upon making decisions during run-time, there will be enormous number of different metrics to be considered. The approach to make run-time management system scale with the growing number of cores is questioned in this thesis:

RQ.4. How can we make run-time management methods scale with the number of cores?

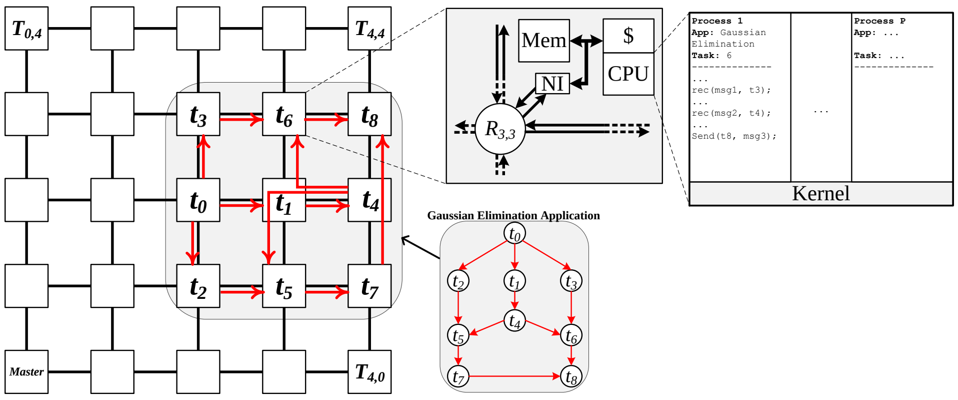
1.4 Summary of Research Contributions

This section briefly explains the main contributions that this thesis makes. These contributions provide novel system- and architecture-level answers in addressing different challenges of many-core systems design, discussed in Chapter 1.2. As aforementioned, the main focus of this thesis is the communication fabric— i.e. the network-on-chip (NoC), and the dynamic nature of many-cores; i.e. the running set of applications changes over time with no prior knowledge, while NoC components and cores may fail randomly during run-time.

This dissertation addresses different communication-centric challenges of dynamic many-core architectures in three major directions. First, based on the observation that in conventional systems kernel SW imposes a significant overhead in handling the transport-layer in communication between parallel tasks, we propose a NI architecture design that operates in transport layer (Tra-NI). Our method moves a significant portion of communication handling from kernel software to NI hardware, where improves the communication performance several folds. Second, we highlight the importance of, and propose several algorithms for, run-time task mapping in many-cores. Through different proposed algorithms, we address our different challenges of interest, mainly in NoC fabric of many-cores. Third, based on the observation that system-level mechanisms (including our mapping algorithm) are not sufficient for guaranteeing a reliable communication, we propose Maze-Routing, a routing algorithm that tolerates run-time permanent failures of NoC components with a set of unique features.

System Model: The contributions of this dissertation are well tailored for a many-core architecture with following characteristics[3], also depicted in Fig. 7. Both applications and PEs of the system communicate and synchronize through message-passing model. That said, applications are composed of parallel tasks, with their

[3] Except for our last contribution, Maze-Routing algorithm, which is more general.



```
Process 1
App: Gaussian
Elimination
Task: 6
---------------
...
rec(msg1, t3);
...
rec(msg2, t4);
...
Send(t8, msg3);
```

...

```
Process P
App: ...

Task: ...
---------------
```

Figure 7. A 5x5 many-core architectural model of this dissertation. Message-passing communication model is used, where cores have private access to their local memory. A light kernel is running on each core, supporting multitasking.

own private memory, which communicate and synchronize through explicit messages. Similarly, cores have access to their own local memory and connect together through a mesh NoC fabric. Each core has a light kernel in the background that handles the system run-time, and may run several tasks of potentially different applications; i.e. supports multitasking. Moreover, the system is managed by one of the cores, called the master core (usually the $T_{0,0}$ in this dissertation), which interacts with user as well. Application execution requests (with number of required cores) are sent to master, where it maps the parallel tasks onto the available cores. Note that our system model is derived from real platforms, namely Intel single-chip cloud computer (SCC) [54] and HeMPS [18] multi-processor system-on-chip (MPSoC).

1.4.1 Transport Layer Aware Network Interface

In the first contribution of this dissertation, we identified the detrimental overhead of protocol handling of transport layer in (kernel) software, and hence proposed our transport layer NI architecture design. In conventional networks, the communication fabric (i.e. NoC in many-core architectures) works in network layer, according to stack protocol of ISO-OSI reference model [88]. In other words, it routes packets from source nodes to destination nodes. When packets reach their destination, the kernel is in charge of bookkeeping and establishing end-to-end communication between parallel tasks. As studied in [12] for computer networks, this involves several context switching and data copying which imposes significant overhead (hundreds of clock cycles, in best cases). This overhead is tolerable in computer networks due to the high latency of the network layer (in order of milliseconds) and its offered flexibility [95]. However, in many-core microprocessors, where the NoC latency is around tens of clock cycles, the kernel overhead in the end-to-end communication latency severely limits the speedup of parallel applications. This problem, as will be discussed in Chapter 2.1, exists in both HeMPS MPSoC and Intel SCC platforms.

To reduce the kernel overhead, our Tra-NI design [38] depacketizes, stores, and retrieves transport-layer messages in the hardware level. According to our experiments, using Tra-NI, the kernel is interrupted around 50% times less (which translates to less context-switching), and most of its work in packet handling is moved to the hardware; i.e. communication is done faster while more CPU time is utilized by actual tasks rather than the kernel. As a result, the network latency is decreased by more than 4 times, and the end-to-end communication throughput is increased by up to 4.7 times. We discuss the conventional kernel overhead and our Tra-NI design, its functionality and benefits in Chapter 2 in more details.

1.4.2 Run-Time Application Mapping

The run-time application mapping procedure of the master core tries to map tasks of a requested application, in an optimal way, onto a set of system PEs that is determined by the run-time application mapping algorithm. Due to system dynamics, the set of available PEs changes over time and the algorithm output for a given application cannot be thought in design-time; i.e. the mapping algorithm needs to decide during run-time and upon arrival of application execution request. This dissertation dedicates a large portion of its effort in exploring potentials of different run-time mapping algorithms. We propose different algorithms that significantly reduce NoC dynamic power dissipation and improve the system performance. Also, we develop algorithms to tolerate run-time permanent failures of NoC components at system level or mitigate performance penalties of online testing methods. In most of our proposed algorithms, scalability over system size is of main concern. Following, we briefly describe different proposed run-time application mapping algorithms, and main obtained results in addressing different challenges. Later in Chapter 3, we explain each contribution in more details with wider results, and critically discuss them.

As shown in equations (3) and (4), the NoC power is a function of both packet length and path length. Hence, a run-time mapping algorithm should place tasks with heavier communication (i.e. tasks with higher $w_{i,j}$ values) in closer PEs to reduce the dissipated NoC power. Accordingly, a contiguous set of PEs is preferred for tasks of an application, as also studied in literature [24]. One of the main contributions of this dissertation is identifying the importance of *first node* selection in obtaining a contiguous mapping [39]. A proper *first node* selection algorithm selects a PE in which there are *just enough* number of PEs around it for tasks of the application. Accordingly, the application will be mapped on a contiguous set of PEs while the remaining PEs are not fragmented. We use our task mapping algorithm [35] after which the *first node* is selected using our agile algorithm [39], namely SHiC. Compared to state-of-the-art [24], we achieve up to 33% and 25% reduction in power and average latency of NoC fabric, respectively. Also, as a scalability analysis, when increasing the many-core size from 8x8 to 20x20 cores, our method results in almost constant average latency and NoC power dissipation per application.

Contiguous mapping places communicating tasks in close proximity which reduces the network congestion. Consequently, the average latency of packets is decreased as our results demonstrated. Also, contiguous mapping relatively isolates traffic of different applications and reduces their interference from execution of each other. In turn, applications will experience a more balanced latency with significantly decreased worst-case latency.

In our next contribution, we utilize the aforementioned benefits of contiguous mapping in average latency reduction. Given the dynamic workload of the many-cores, it is not always possible to map an application onto a contiguous set of PEs. A non-contiguous mapping increases the communication latency and hence execution time of running applications. On the other hand, limiting the system to only contiguous mappings keeps applications wait for contiguous PEs to free up, reducing the overall system throughput. Our contiguity adjustable mapping algorithm [40], namely CASqA, defines an α parameter ($0 \leq \alpha \leq 1$) in which determines the level of allowed dispersion (non-contentiousness) in PE selection. Strictly contiguous mapping is determined by $\alpha = 0.0$, while $\alpha = 1.0$ determines conventional non-contiguous mapping. Surprisingly, results demonstrate that partially contiguous mapping ($0 < \alpha < 1$) can even increase the system throughput over the conventional non-contiguous mapping. Similarly, with smaller α values, up to 35% saving in NoC power dissipation can be achieved, while maintaining the same throughput as non-contiguous mapping.

We utilize balanced latency benefits of contiguous mapping in our next contribution. As shown in Fig. 8, compared to non-contiguous mapping in CASqA algorithm, different NoC latency metrics are significantly smaller for $\alpha = 0.0$. Particularly, the worst-case latency of the network is more than 10 times smaller when restricting to only contiguous mappings. We use this insight to offer a better QoS to real-time applications [42]. Once a real-time application is requested for execution, instead of dispersing the selected set of PEs, we use the multitasking capabilities of each PE to keep the mapping result contiguous. Consequently, packets of the real-time application are delivered with a more balanced latency values. For instance, results show over 50% improvement in worst-case latency and up to 33% times improvement in deadline misses for MPEG-4 video stream coding application [82].

So far, our run-time mapping algorithms were concerned about NoC power dissipation, application execution time, and the scalability of methods over system size. As a step further in contributions of this thesis, we explore potentials of run-time mapping algorithms in reliability assurance of many-cores. Online testing methods are required to ensure correct functioning of system components and to detect when they go faulty. Once a component goes under test, it becomes unavailable until it is

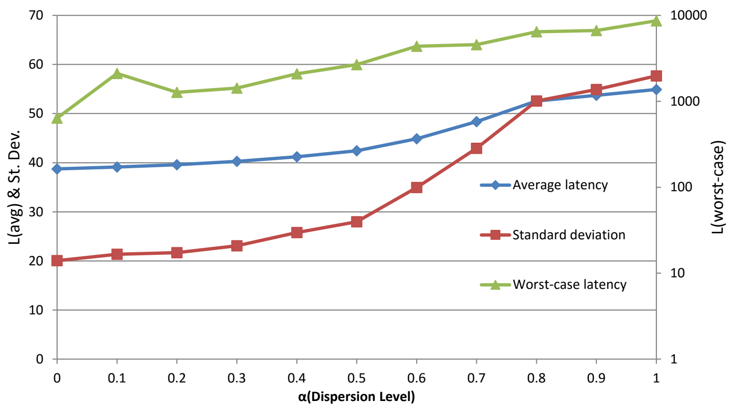


Figure 8. Different NoC latency metrics of CASqA [40] algorithm for different α (allowed dispersion level) values [42].

tested and continuation of its correct functioning is validated. Cores need to be tested after they are being utilized more than a specific threshold; e.g. 10M instructions are executed. Though a core might pass its testing threshold, we might not be able to schedule online testing on it as it is busy executing an application task and/or there might not be enough power budget for testing at that moment. Note that test methods are power hungry as they active all modules of a component in short period of time. Accordingly, at a given time, there might be several cores to be tested, while we cannot test all due to power budget limits. On the other hand, testing a core releases it for further deployment by application tasks. Hence, testing cores that can benefit execution of future applications is preferred. Because of the preference of a contiguous application mapping, in our next contribution [49], we propose a test-aware mapping and test scheduling algorithm which frees up contiguous regions of cores for future applications. Results demonstrate competitive throughput of the system compared to many-cores without testing.

As our last contribution in run-time mapping algorithms, instead of architectural methods, we propose to tolerate run-time failures of NoC components at system-level. Accordingly, our SHiFA [41] approach maps applications onto PEs such that their communication will not require the use of faulty elements. To novelties are utilized to make SHiFA scalable and increase accessible PEs of the system. First, the system master has mobility features, called mobile master (MM); i.e. it can migrate from one PE to another. Also, SHiFA works in a hierarchical manner, where the actual application mapping is done by application managers (AMs). A high-level representation of SHiFA hierarchy and its fault-aware application mapping is demonstrated in Fig. 9 (a) and (b), respectively. As can be seen, all the tasks of the application are mapped onto PEs accessible by AM and none of the communications go through the crossed faulty node. When up to 8 links are broken and by utilizing a

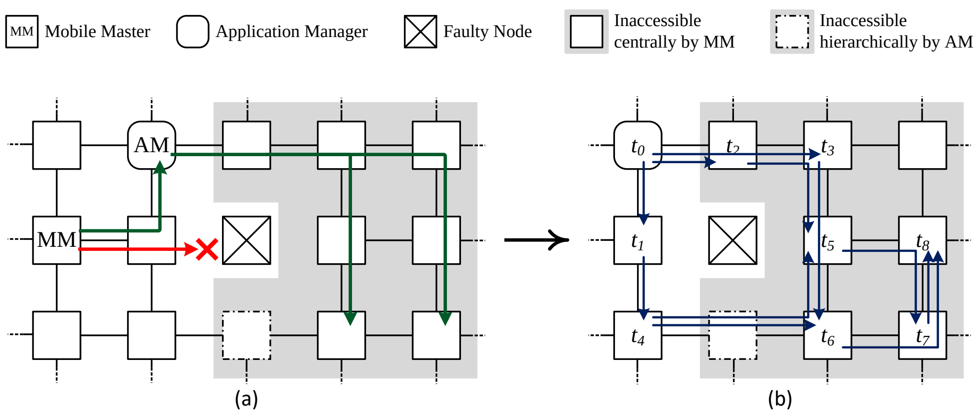


Figure 9. SHiFA demonstration: (a) using XY routing and within the same architectural limitations, 7 nodes can be accessed by AM out of 8 nodes that are not accessible by MM. (b) A feasible mapping solution of Gaussian Elimination application. [41]

simple routing algorithm, results show 100% accessibility of PEs and 99.41% of successful mapping. Also, different scalability analysis show a significant improvement compared to the state-of-the-art approaches.

1.4.3 Fault-Tolerant Routing

In the last contribution of this thesis, we propose a routing algorithm to tolerates faults in NoC components [36], namely Maze-routing algorithm. Maze-routing is the first algorithm that provides four important properties at the same time. First, it is fully-distributed where no centralized component is utilized. Second, it provides guaranteed delivery (i.e. full fault coverage), which is to guarantee to deliver packets when a path exists between nodes, or otherwise indicate that destination is unreachable, while the deadlock and livelock are avoided. Moreover, it imposes low area cost as well as low reconfiguration overhead upon new faults. The main motivation behind this work is that in our previous contribution (SHiFA approach [41]), we observed that a system-level only method cannot be sufficient as they can lead to deadlocks.

Briefly describing, our Maze-routing algorithm works as follows. A packet can be in either normal or traversal modes. In normal mode, each router forwards the packet to a productive output port[4], as long as possible. When the packet enters a router with faulty components such that a productive output port does not exist, the packet goes to traversal mode. In this mode, the packet traverses around the faulty region hop by hop until it enters a node where it is safe to revert back to normal mode. This procedure continues until the packet reaches its destination or one of the

[4] A productive output port is one that moves the packet closer to its destination

Table 1. Validation environments used for different contributions of the thesis.

Methodology	**Description**
FPGA prototype	HeMPS-based full system stack with a 5x5 mesh
In-house simulator	Message-passing transaction-level application models with Noxim-based cycle-accurate NoC
NOCulator	Cycle-accurate NoC with shared memory trace-based application models

routers it visits detects that the destination is unreachable. We formally prove that our method provides guaranteed delivery.

Our evaluations show that Maze-routing has 16X less area overhead than other algorithms that provide guaranteed delivery. Moreover, it delivers high performance in presence of faults. When up to 5 links are broken, it provides 50% higher saturation throughput compared to the state-of-the-art.

1.5 Methodology

Building real many-core systems will be very costly; both money-wise and time-wise. Hence, we need to use simulators to validate new ideas and algorithms quickly and cheaply. As shown in table 1, in this thesis, We use three different simulation environments.

HeMPS MPSoC platform [18] is used in the beginning to validate our NI design, Tra-NI, and run-time application mapping, contiguous neighborhood allocation (CoNA). We made major modifications to HeMPS platform to match it to our research requirements, details of which will be presented in Section 2.5. HeMPS provides a synthesizable full system model of a many-core architecture, which makes the results robust and reliable. However, using full system prototype comes along with its own issues. For instance, as the system is modeled mainly in VHDL, the simulation is very slow, and debugging is extremely difficult. Due to lack of commercial tooling, debugging application-level C code boils down to tracing signal values and absolute memory locations in RTL simulations. Also, exploration of new ideas are usually challenged by the practical limitations of the system, like how much memory or how many cores the FPGA could fit. These technical limits make the research extremely challenging and laborious, with usually limited value add to the research topic.

Limitations of a real system prototype motivated us to develop our own in-house System-C simulator. For the network-on-chip model, we took the Noxim [81; 21] simulator and pruned the code to run faster the specific use-cases of this thesis. Then we added the PE models, similar to the HeMPS platform. The PEs have their own local memory, and can execute message-passing applications in transaction level.

Tasks, running on PEs, send and receive messages to/from the network, and spend clock cycles in between. As our run-time mapping algorithms are validated on this platform, we elaborate it more in Chapter 3.

The last contribution of this thesis (Maze-routing algorithm) is based on deflection-based [75; 33] routing methods, which is not supported by Noxim. Consequently, we validated our contribution using NOCulator [32; 33] simulator, which supports deflection-based NoC models. As NOCulator, originally, did not support faulty networks, we added the fault models as well as our maze-routng algorithm on top. Unlike HeMPS and our Noxim-based simulation environment, NOCulator is based on shared memory many-core architectures. The NoC model is cycle-accurate, while the PEs model the memory access traces of applications.

1.6 Thesis Organization

This thesis is composed of two parts. First, we briefly explain the conducted research, within 6 chapters. Then, we present the original publications made through this thesis work. It consists of 9 original research papers, published by the author and his collaborators.

Fig. 10 shows the navigation of the first part of this thesis, containing the research questions (RQs) answered in each chapter, included papers, and used methodologies for result validation. Chapter 1 provides a background on the research topic and its design challenges. Also, it proposes the research questions that this thesis is aimed at investigating them, and briefs the contributions of this thesis. Chapter 2 explains our transport layer aware network interface design, Tra-NI, and the obtained results. Chapter 3 explores different contributions of this thesis in run-time application mapping of many-core processors. Chapter 4 elaborates our unique fault-tolerant routing algorithm for mesh NoCs. Chapter 5 summarizes the original publications included in this thesis, describes the author's contribution. Finally, Chapter 6 concludes and discusses this thesis. It summarizes our research contributions, achievements, and limitations. Moreover, it points out few areas, where future research and study is needed.

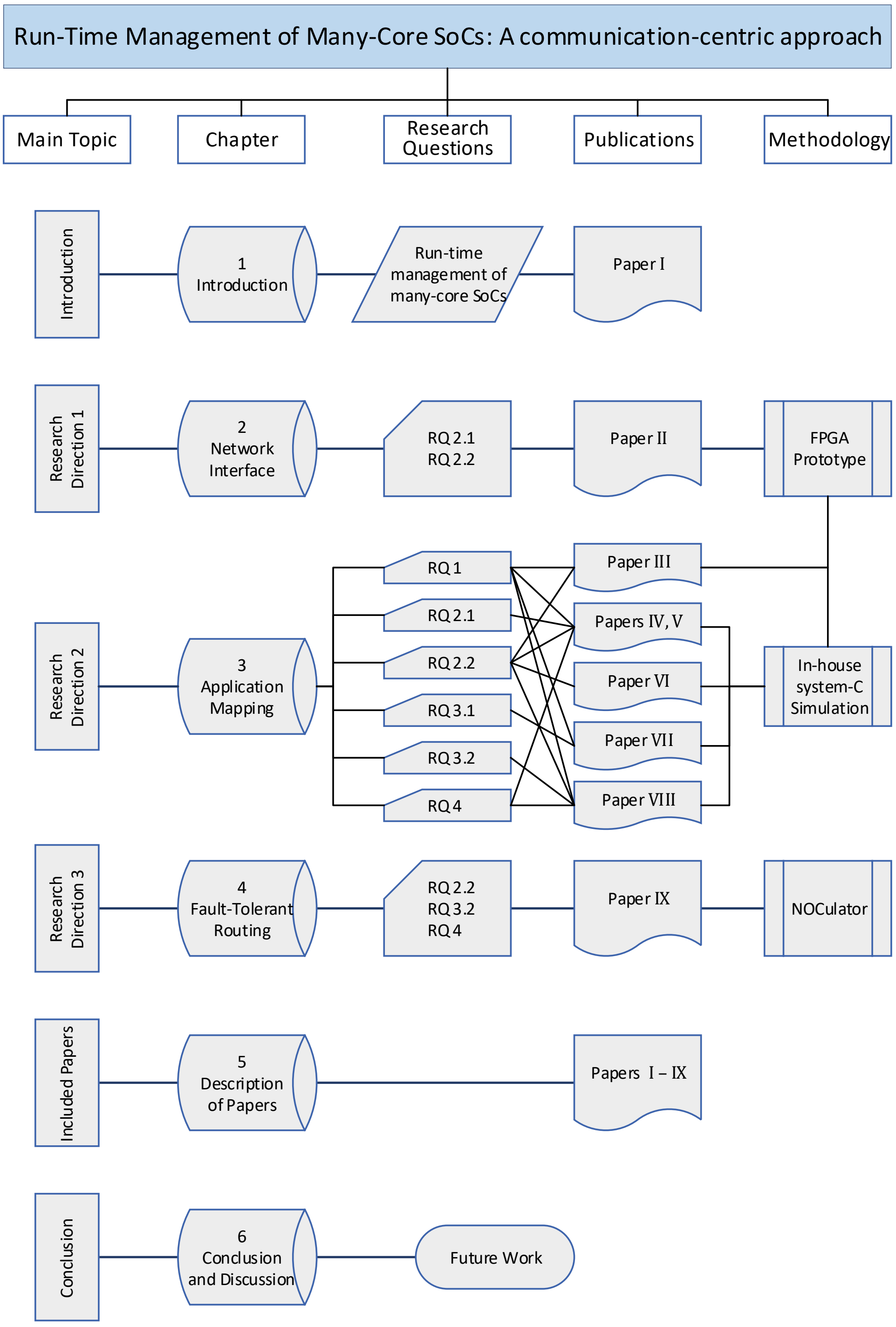


Figure 10. Navigation of the Thesis.

2 Transport Layer Aware Network Interface

In the message passing communication model, tasks communicate using explicit messages. The two fundamental functions in the message passing model are `send` and `receive` primitives. To keep the underlying hardware transparent to the application design, tasks send and receive data using their task ID as the sender and receiving addresses. These application level messages need to be transformed into formats, understandable by physical routers and links; e.g. a set of flow control digits (flits). According to ISO-OSI standard model [88], this is done through two layers of translation. First, the transport layer takes the application data, encapsulates it in a segment and generates network packets. Network packets are then translated into sequence of flits in the network layer, routed towards destination with NoC routers and links. Conventionally, the transport layer is implemented by operating system (OS) software, while the network layer is implemented by network interfaces (NIs) hardware.

In this chapter, as the first contribution of this dissertation, we investigate the OS overhead in implementing the transport layer in many-core architectures. We propose a network interface (NI) architecture (Tra-NI) with understandings about the transport layer, in which, the OS software overhead is offloaded to the NI hardware [38]. More details about this contribution, and obtained results can be found in II

2.1 Related Works

As the name represents, network interfaces (NIs) interface between two sides of a network that have different communication protocol; i.e. they establish a conversation between two parties who do not speak the same language. Accordingly, existing NIs can be categorized based on the two protocol ends they work with. In the following, we review some of existing NI designs, proposed for different protocols and scenarios of NoC architectures.

Several network interfaces are proposed that work in the physical layer of globally-asynchronous locally-synchronous (GALS) systems [68; 76; 79; 98]. They deal with meta-stability issues when signals are sent/received to/from the NoC, which is asynchronous or operating in a different frequency than cores.

A NI design for serial NoCs is proposed in [65], which works in several layers and converts serial bit-stream of the network to data packets of the system cores and back. It (de-)packetizes the data (network layer), offers error detection and retransmission (data link-layer) and scrambles the bit-stream (physical layer).

Kim et. al, propose NIUGAP [62], a NI which reorders packets that arrive out-of-order in the network interface. It uses gray codes to efficiently synchronizes the packet delivery in the network layer. An AXI compliant NI is designed in [28] that reorders the memory accesses that are delivered (to PEs) out-of-order.

Several NIs are proposed for different master/slave shared-memory transaction models [67; 8; 44], such as OCP, AXI, and/or DTL. They interface between the read/write transactions issued by cores and the packet transmission happening in the network. Radulescu et al., [84] proposed a NI which offers guaranteed service. Their work supports existing bus protocols such as AXI, OCP, and DTL. Similarly, Attia et. al [8] designed a NI compliant with OCP. Master and slave type NIs are used to connect master/slave intellectual property (IP) cores to NoC. To reduce area overhead, several IPs of the same type can connect to one NI where buffer space is shared between them. In another design [25], they utilize stoppable clock techniques to reduce the power consumption of their NI. Another OCP compliant NI which shares the buffer space between several master/slave cores is presented in [44]. In [91], authors propose a NI with worst-case guarantees. Their work eliminates the need for buffers and credit based flow control, which significantly improves the area footprint of their work.

2.2 Transport Layer Functionality

In the message passing communication model, tasks communicate using explicit messages. For instance, message passing interface (MPI) standardizes several key functions for communication between parallel tasks in the message passing model [102]. Nonetheless, functions `send` and `receive` are the two fundamental primitives in the message passing model. In its simplest form, each task of a parallel application has a unique identification number (task ID), where the communication is addressed through it.

Fig. 11 shows a pseudo-code for tasks 3 and 4 (t_3 and t_4) of the Gaussian elimination application (see Fig. 5). As can be seen, tasks communicate with each other through their task ID number, while do not now about the physical core in which tasks are mapped onto. In other words, the transport layer keeps the networking and architectural details transparent to running tasks. In the following we present a very high level description of the transport layer services in establishing communication between two tasks. Note that while there are more services associated with the transport layer, they fall beyond the concerns of this dissertation.

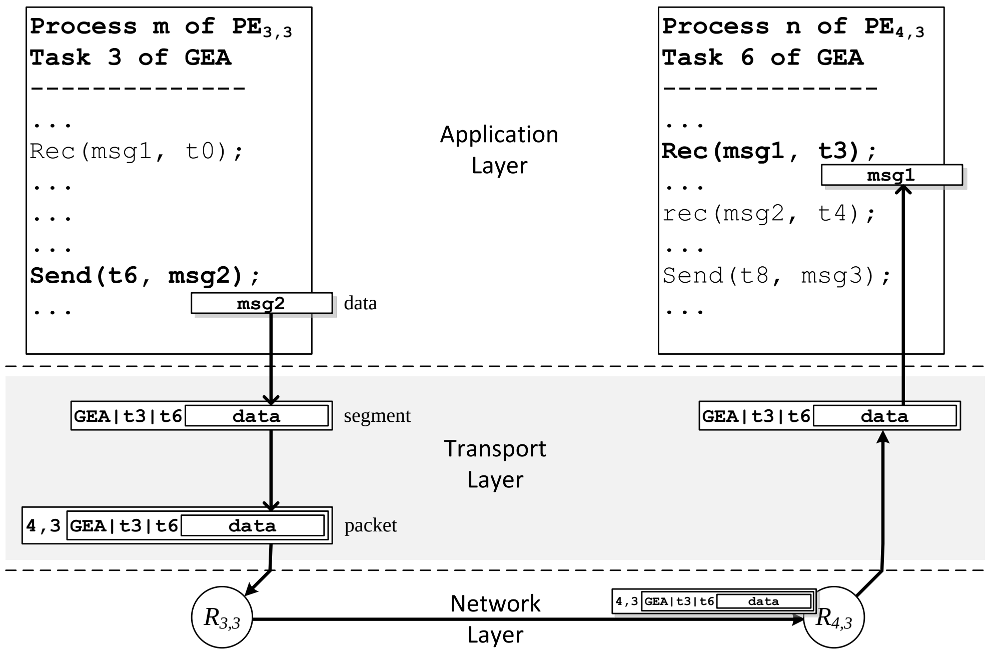


Figure 11. Different layers in communication between tasks.

Sender side: Once a source task wants to send a piece of data to a destination task, the transport layer service encapsulates the data into a message unit, called data segment. Then, the data segment is attached to the network packet addressed to the physical address of the receiving task; i.e. the core in which the destination task is mapped onto. This is shown in the left side of Fig. 11 for task 3 of Gaussian elimination application (GEA) mapped onto $PE_{3,3}$.

Receiver side: On the other side of the communication, the transport layer demultiplexes the incoming data segments and passes the encapsulated data to the correct communication streams. Right side of Fig. 11 shows this procedure for task 6 of GEA running on $PE_{4,3}$. It looks up the incoming messages and passes the message from the appropriate application and task ID to the `receive` function. Also, the task execution is suspended in case the requested message is not yet arrived, and is resumed after message arrival. Moreover, due to existing multitasking, the transport layer service is responsible for correct sharing of access to the network hardware.

2.3 Kernel Overhead

In conventional systems, the transport layer is implemented by the operating system [96]. Since the network layer is handled by hardware and the transport layer by

OS software, the transport layer significantly penalizes the communication between two tasks [95]. This becomes more of an issue in on-chip many-core architectures because of two main reasons. First, the network delay is much smaller than that of computer networks. Moreover, the on-chip communication is one of the main elements in exploiting the existing parallelism of many-core architectures. In the following, we describe the main kernel overheads in providing the transport layer services.

Sender side: The `send` function makes a system call and causes a context switch to the kernel mode; with penalty of $P_{context}$ associated with it. The context switch is necessary as the kernel and not the program code knows about the physical PE of the destination task, and to manage concurrent accesses to the network. The kernel packetizes the data (data→segment→packet– P_{pack}), and injects it out to the network ($P_{mem \to NI}$).

Receiver side: Same as sending messages, the `receive` function makes a system call with $P_{context}$ penalty. The kernel searches its buffer space (P_{search}) for the requested message, with different consequences based on the search result, as follows. If the requested packet is already delivered, it has made a hardware interrupt on its arrival ($P_{context}$), and the kernel has depacketized (P_{depack}) the packet, picked a buffer space to place the segment (since no task has been awaiting for the packet at that time– P_{place}), and performed the data copy ($P_{NI \to mem}$). Now the requested data is copied from the kernel buffer to task memory and the task execution is resumed ($P_{mem \to mem}$). Otherwise, the task is suspended. Once the requested message arrives, a hardware interrupt is made ($P_{context}$), the segment is depacketized (P_{depack}) and since the task is waiting for that message, the data is directly copied to the task memory ($P_{mem \to NI}$), bypassing the kernel buffers. Finally, the execution of the suspended task is resumed. Note that the depacketizing and search penalties are taken into account, because a simple FIFO cannot be utilized for the kernel buffer. Packets are not necessarily requested in the same order as they arrive and there can be several packets with the same segment header; i.e. the kernel needs to give packets an ordering number.

2.4 Tra-NI Design

As can be intuitively derived, the kernel overhead in the receiver side is significantly larger than that in the sender side, especially when a message is arrived before it is requested by `receive` function. Accordingly, we design a transport layer aware network interface (Tra-NI), which the heaviest kernel functionalities in the receiver side are performed in hardware. Fig. 12 shows the block diagram of our proposed design.

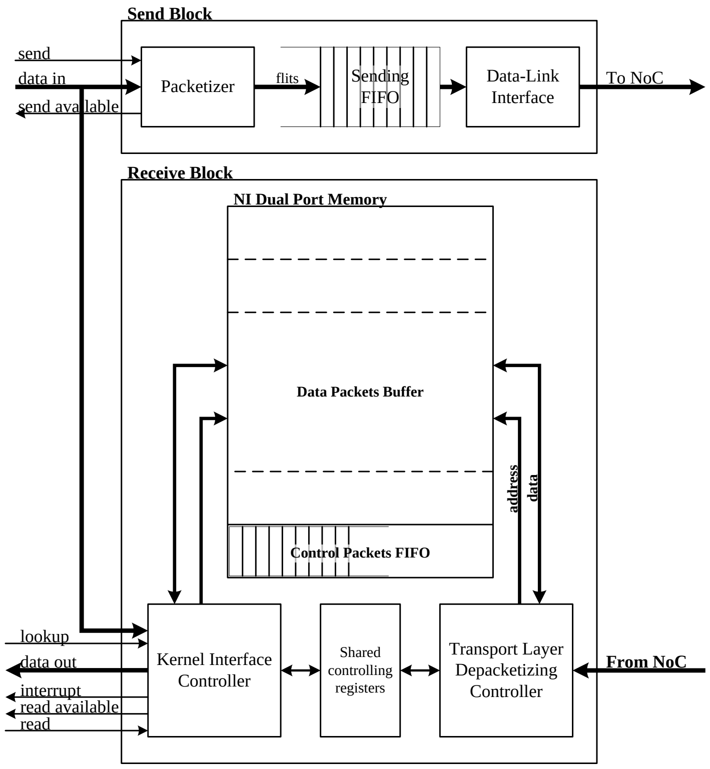


Figure 12. Tra-NI block diagram, redrawn from [38].

As can be seen in Fig. 12, instead of the kernel, Tra-NI holds the buffer for storing incoming messages. When `receive` function is called, the kernel issues a lookup, where Tra-NI searches its buffer, and returns the requested packet if found, otherwise it informs the kernel about the lookup failure and stores the pending request. Once a packet of a pending request arrives at Tra-NI, it interrupts the kernel to copy the packet and resume the execution of the suspended requesting task. As a result, our design moves P_{depack}, P_{place}, and P_{search} to Tra-NI hardware with significant speedup. Moreover, no software based memory to memory copy is performed ($P_{mem \to mem}$) anymore, while one less context switch is occurred when the packet is arrived before it is requested by a task.

2.5 Experimental Setup

HeMPS [18] (version 3.9) is a register-transfer level (RTL) model of a message-passing many-core SoC architecture, which connects plasma processors [85] using HERMES NoC [74]. Each core runs a light OS to support multitasking and communication between tasks. The HeMPS Generator framework [73] generates synthesizeble RTL model using VHDL well tailored for field-programmable gate array (FPGA) synthesis. HeMPS platform is very similar to our system model, described in Fig. 7. In addition to the local memory, CPU, and NI, each PE also includes a direct memory access (DMA) module to speed up data copy between NI and local memory. The system master loads the binary code of each running task from its connected application repository and sends it to plasma cores for execution.

As mentioned above, each core of the HeMPS version 3.9 platform supports multitasking. Even so, it does not support random execution requests during run-time nor multiple executions of the same application. Structure of the application repository as well as the C code of the kernels in both master and other cores are modified to enable these essential features needed for different contributions of this dissertations.

Extending HeMPS to support run-time execution of independent applications, different related work in run-time application mapping algorithms are implemented in kernel code of the master core. Alongside, we realized the performance-killing bottleneck of protocol handling in plasma kernels, which fades away any potential benefit of different mapping algorithms. Accordingly, this led us to the first contribution of this thesis, Tra-NI [38], to improve the execution time of applications by improving their communication speed. We synthesized a 5x5 prototype of our modified HeMPS platform on Xilinx Virtex-6 FPGAs. As reported in [35], some parts of results are extracted using the FPGA prototype.

2.6 Results

As explained in Chapter 2.5, we used HeMPS [18] platform to validate our NI design, while kernels are modified to support run-time application execution requests and multiple executions of the same application. Tra-NI is developed in VHDL and utilized in a 6x6 network operating at 100 MHZ.

Performance: Two showcase applications are developed to measure the main performance improvements obtained using our Tra-NI. An application, called *flow*, where a source task constantly sends messages to a sink task; and a *ping-pong* application where a message is ping ponged between them. The *flow* represents the producer consumer type of applications, while *ping-pong* is for collaborative applications. In relation to conventional NI design, our results represent 4.78 times higher bandwidth between source and sink tasks, and 39% bandwidth improvement for the ping-pong application. In other words, using Tra-NI, the ping-pong application can inject its data with rate of up to 0.19 flits/cycle. As reported in [70], this value is only 0.004 flits/cycle when running ping-pong application on Intel SCC platform. In both experiments, the exchanged data chunks are 128 flits long. This 50 times higher injection rate is obtained while the Intel SCC platform cannot support multitasking. More detailed results can be found in our original publication [38].

3 Run-Time Application Mapping

In the second direction of our contributions, we deal with run-time resource management of many-core architectures. In particular, we propose different algorithms on how to map tasks of a parallel application onto cores that are currently available. We study that how the mapping algorithm can be utilized to tackle different challenges of many-core architectures.

Application mapping: As mentioned in Chapter 1.1.2, a message-passing application (such as GEA shown in Fig. 5) is often represented as a directed task graph $A_p = TG(T, E)$. T is the set of tasks ($t_i \in T$) and E is the set of edges ($e_{i,j} \in E$), representing data communication from t_i to t_j. We denote the number of tasks ($|T|$) as the application size, which is the number of available PE required for its execution. Similarly, we use an architecture graph $AG(N, L)$ to model our many-core system shown in Fig. 7. Nodes ($n_{x,y} \in N$) are composed of a router and a PE, connected together through on-chip links ($l_{n_a,n_b} \in L$). Due to the dynamic workload, at any given time, a changing subset of nodes ($N' \subseteq N$) will be available for running new applications. Accordingly, the run-time application mapping can be defined as a mapping function from set of application tasks (T) onto the available set of nodes (N') of the moment:

$$\begin{aligned} &function\, map : T \rightarrow N'\, defined\, as \\ &\qquad map(t_i) = n_{x,y} \;\; s.t.\;\; \forall t_i \in T, \exists n_{x,y} \in N' \end{aligned} \tag{5}$$

For the sake of simplicity and without loss of generality, in our proposed mapping algorithms we assume that once a core is allocated to a task, it becomes unavailable. Hence, once an application is mapped then $N' \leftarrow N' - map(A_p)$, and once leaving $N' \leftarrow N' + map(A_p)$. In the general case of multitasking per core, a core can be assumed available if there are enough CPU and networking resources left for a new task.

Congestion: During execution of different applications, packets will content for the NoC resources, causing network congestion. The network congestion significantly increases the packets latency and limits the maximum achievable throughput [27]. From our application mapping perspective, we define two types of congestion: external and internal. External (a.k.a. inter-application) congestion occurs when packets of different applications content for network resources. Whereas, internal (a.k.a. intra-application) congestion is due to contention between packets of the same application.

3.1 Related Works

There are enormous number of works proposed in literature to cope with spatial mapping of applications onto system cores. Most of these works deal with static application mapping for multi-core embedded systems where the set of running applications are known and fixed in design-time. On the other hand, several works, like us, deal with dynamic workloads of many-core systems, where the set of running applications is unknown beforehand and changes during run-time. In this section, we explore different existing works of the latter group. Each of these works have different optimization metrics and try to achieve different goals.

Carvalho et al. [19; 20] propose a variety of different mapping algorithms; such as nearest neighbor (NN), path load (PL) and best neighbor (BN). The main objective of the proposed algorithms is to reduce network congestion and accordingly improve applications execution time. The proposed algorithms are compared against a naive first free (FF) algorithm, which is to map the application tasks onto the first available cores regardless of the application task graph and with no optimization metric. These works use a clustering mechanism to map the first task of an application. The first task is mapped onto the closest core to the system master among a set of sparse cores (cluster nodes); i.e. an application is mapped only when a cluster node is available. In NN algorithm, each task of an application is mapped onto the nearest available neighbor of its communicating task, regardless of the communication volumes ($w_{i,j}$ values in Fig. 5). Accordingly, NN tries to reduce network congestion by generally placing communicating tasks in closest neighborhood. The PL algorithm, on the other hand, examines all the available cores and maps a task onto the core in which the total channel loads of the communicating tasks is minimized. Though PL achieves better results than NN, it is computationally intensive. Accordingly, BN works like NN, where the PL algorithm is used to select a core among the nearest neighbors.

The run-time mapping algorithm is divided into two steps in [24]; region selection, and task allocation. In the region selection step, they try to find a set of cores such that it is contiguous and makes the least fragmentation in the remaining set of available cores. The closest available core to the system master is selected as the first node in the region, and the region is formed around this core. Afterward, the application tasks are allocated to the selected region of cores aiming at reducing NoC power, by allocating the tasks with higher communication volumes first. Later, the user behavior is incorporated into their mapping algorithm [22]. Applications are tagged as either critical or non-critical, using tree-based model learning. Congestion reduction is the main goal in mapping of critical applications, while non-critical applications are mapped by their conventional algorithm.

A heuristic for run-time application mapping in heterogeneous MPSoCs with FPGA tiles is proposed in [77]. The algorithm, implemented in three steps, aims at reducing the on-chip communication power. In the first step, tasks are prioritized based on their computation and communication demands. Then, available cores are prioritized based on their load. Accordingly, the most prior task is mapped onto the most prior core. The two last steps are done for all tasks, and backtracking is used if no solution found. This work implies a high computation complexity due to the backtracking, and use of complex optimization target functions.

Asadinia et al. [7] reduced network congestion, using their virtual point-to-point (VIP) connections [71]. Accordingly, when two communicating tasks are not placed on neighboring nodes, a VIP connection is established between them. This work, however, relies on the customized VIP design, where they can easily become fully occupied depending on the application task graph.

Some researchers, on the other hand, proposed decentralized algorithms to achieve a better scalability over the number of cores. ADAM [34] is an agent-based approach to distributed run-time application mapping of many-core systems. Cluster and global agents of the system, which are distributed over the whole system, use a negotiation policy to implement the approach. The high complexity of the negotiation policy makes ADAM suitable for many-core systems with beyond 100 cores.

Another decentralized mapping algorithm is proposed in [105] for tree-structured task graphs. In addition to tasks, the algorithm maps edges of a given application; i.e. the routing algorithm for the application. The central master maps the root task, where other tasks are mapped by their parents; given the tree-structure of task graph. The work tries to minimize the load on network links and the distance between communication tasks to achieve better latency results. Authors, however, do not address their approach in monitoring the load put on network links. STDM [53] and DistRM [63] are other decentralized methods, which try to improve over centralized and/or other decentralized methods. They start application mapping from a random node and explore neighboring nodes to collect enough resources. In case of unavailability of resources around the randomly selected node, DistRM explores distant nodes as well, while STDM selects a new random node until several times.

Network power and latency optimization is the main objective of the above mentioned works, where a regular mesh NoC is considered. Whereas, in recent years, run-time application mapping has gained attraction to achieve goals other than NoC metrics optimization.

As a fault-tolerant run-time application mapping, FARM [23] approach considers failures in PEs (but not in NoC links and routers). Accordingly, failed PEs are omitted from application mapping. Also, once a PE breaks while running an application, spare-cores are used to migrate running tasks them.

Run-time application mapping is used to improve the thermal profile of many-core systems [30; 58; 61]. The main goal of these works is to achieve a higher system utilization, while preventing thermal violation of the system. Accordingly, cores voltage and frequency are adjusted and applications are mapped and migrated such that the same amount of job is done while leading to a lower chip temperature. The decreased temperature compared to the peak safe temperature leaves room for additional system utilization.

Variation, reliability and aging of transistors are also considered in several works [48; 59; 60; 89]. These methods consider variation in their algorithm to achieve better system utilization within a smaller power budget. Also, different mapping techniques are utilized to improve system reliability and slow down its aging over time, with minimal sacrifice in the system performance.

Worth noting, as mentioned in Chapter 1.4.2, our findings in optimizing the NoC metrics show that a contiguous application mapping is significantly beneficial. However, as described by some of recent works [48; 58; 89], seeking for a contiguous mapping might not be enough for thermal and/or aging optimization of many-core systems. Moreover, although the mentioned works may favor/search for a contiguous mapping, they have no limit on the contiguity of the solutions and allow any level of dispersion.

3.2 Validation Environment

As mentioned in 2.5, technical challenges of RTL full-system simulation motivated us to develop our own System-C simulator. We use this simulation environment to validate our different run-time mapping algorithm proposals. The simulator maps applications, described by task graphs, onto system PEs and extracts different run-time metrics per application execution, such as execution time, network latencies, power dissipation per application and so on.

3.2.1 In-House System-C Simulation

We selected the Noxim simulator [81; 21] to model the the communication architecture of our many-core system model. Noxim is a cycle accurate network-on-chip simulator developed in System-C. It provides a very detailed model of NoC architecture and can be configured for a variety of different parameters; e.g. network dimensions, routing algorithm, selection strategy, buffers depths, etc.

To model the PEs of our system model (Fig. 7), we replaced Noxim PEs with models of our own. In Noxim, each processing element can only generate synthetic traffic in a specific rate and in accordance to a traffic pattern. Whereas, in our system model, processing elements execute message-passing applications where tasks running on each CPU receive data from and send it to other tasks according to the associated task graph. Accordingly, each processing element of our simulator runs in two modes: user mode and kernel mode. In the user mode, the behavior of the allocated tasks are emulated, where each task is composed of three main primitives, `Compute N`, `Send` t_i `N`, and `Receive` t_i. `Compute` basically elapses `N` clock cycles as a high-level model of performing calculations. `Send` sends a data packet of length `N` to task t_i. Similarly, `Receive` waits until a message is received from t_i. Moreover, there is a `Loop N` primitive, which repeats the same behavior for `N` times. In the kernel mode, on the other hand, the processing element is in charge of multitasking, inter-task communication, and communicating with master PE.

Accordingly, the following scenario happens once an application is going to be executed on our many-core simulator. The system master maps application tasks on the system PEs according to their availability using the active mapping algorithm. Once the target PE of each task is determined, the system master sends the behavioral model of each task along with the mapping result of the application to the target PE. The kernel of target PEs load their task model into their local memory, and pass the CPU to running tasks. Once a task needs to send or receive some data, the control is passed to the kernel mode, where the logical task addresses (t_i) are translated to physical PE addresses, the intended data is sent/received, and finally the user mode is set back. Once a task finishes its execution, the kernel informs the system master. An application is considered terminated, when all of its tasks are terminated.

3.3 Contiguous Application Mapping

As one of our main contributions in this direction, we elaborate benefits of contiguous application mapping, and propose different solutions to obtain it. Contiguous mapping is to map tasks of an application onto contiguous set of nodes with no fragmentation between them. This reduces the average hop count in packet traversal between communicating tasks; i.e. improves the NoC power dissipation according to equation (3), while isolates the communication of different applications from each other; i.e. reduces the external congestion and thus the average packet latency of the network. As opposed to contiguous mapping, a dispersed application mapping maps tasks of an application onto distant and fragmented set of nodes. This increases the average hop count between tasks of an application, resulting in higher NoC power, and places tasks of different applications between each other, leading to increased external congestion. Accordingly, we prefer to map an application as contiguous as possible. A near-contiguous allocation that not only maps tasks onto neighboring nodes, but also does not lead to fragmentation of remaining available nodes, N'.

3.3.1 Quantifying the Contiguousness

The contiguousness of a a mapped application (A_p) can be measured as the the average pairwise Manhattan distance (MD) between the allocated nodes [10], called mapped region dispersion (MRD):

$$MRD_{map(A_p)} = \frac{\sum_{t_i,t_j \in T} MD(map(t_i), map(t_j))}{\binom{|T|}{2}} \tag{6}$$

The most contiguous set of nodes; i.e. the one with the least MRD value is almost circular [10]. However, due to the mesh topology of the NoC, a circular region leads to irregularity and fragmentation of remaining nodes (N') in the long term. Hence, square regions, as the closest rectangular shape to a circle, would be preferred in our problem at hand. The the MRD of a square with $|T|$ nodes is $MRD_{SQ_{|T|}} = 2 \times \sqrt{|T|}/3$. Accordingly, in order to normalize the MRD so that contiguousness of two applications with different sizes can be compared, we define normalized MRD (NMRD) as:

$$NMRD_{map(A_p)} = 1 + \frac{|MRD_{map(A_p)} - MRD_{SQ_{|T|}}|}{MRD_{SQ_{|T|}}} \tag{7}$$

Accordingly, NMRD defines a normalized metric for evaluating the contiguousness of a mapping result, where $NMRD = 1$ means a strictly-contiguous allocation. To evaluate the representativeness of the NMRD metric, we ran 32000 different applications on a fully loaded 16x16 network. We extracted the applications execution time and the NoC power in accordance to the NMRD of the mapping result. Fig. 13 represents the normalized values of the obtained results. As can be seen, both the NoC power and the execution time of applications increase by the NMRD of their mapping result.

3.3.2 CoNA and SHiC Methods

Related work, generally, emphasizes on *which task of an application* to map first and how to map other tasks in relation to the already mapped tasks; e.g. [24; 19]. As opposed to related work, we introduce and highlight the importance of *first node* selection (instead of *first task* selection) in achieving a contiguous mapping. The mapping algorithms, in general, start from the *first node* and try to allocate required number of nodes from the neighboring area [19; 24; 7; 34; 53; 63; 105]. Accordingly, an optimal *first node* needs to have just enough number of nodes around it to obtain contiguity of the mapped application, while preventing fragmentation in N'.

Finding a contiguous set of nodes is a polynomial problem [29], with complexity of $O(|N|^3)$ where $|N|$ is the number of cores. Since strictly contiguous mapping

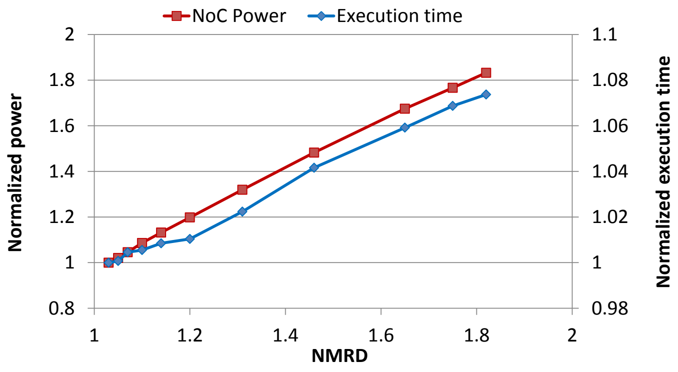


Figure 13. The normalized execution time and NoC power dissipation as a function of the NMRD values.

is not always possible, especially in a fully loaded system, while other aspects such as fragmentation of remaining nodes are of interest; however, the best *first node* selection problem turns into a clustering problem [46] with NP-hard complexity. Indeed, proposed methods need to be scalable, which makes use of heuristics with faster but approximate resolution an inevitable choice.

We first propose CoNA algorithm [35], wherein, the *first node* is the nearest one to system master, among the ones with the highest number of available immediate neighbors[1]. This ensures that the selected *first node* has a minimum number of available nodes and some level of contiguity is attained. Besides the *first node* selection, the CoNA algorithm deals with placement of other tasks of an application. It assume the task graph to be undirected and traverses it in the breadth-first order, beginning from the *first task*. Since *first node* has the maximum number of available neighbors, to further avoid congestion, *first task* is the one with the highest number of connections to other tasks. Accordingly, each task, which is met through a task preceding it (called parent), is placed on the closest node to its parent. If more than one closest node exists, the one fitting into a smaller square with *first node* is picked. A high level pseudo code of CoNA is described in Alg. 1.

Although CoNA improves the contiguousness of mapping over related work, its *first node* selection algorithm lacks an important point. An optimal *first node* is not only a function of the current configuration of the system (reflected as N'), but also a function of the incoming application. For instance, as shown in Fig. 14 (a), CoNA selects $n_{1,1}$ as the *first node* despite the application size. While this results in the least fragmentation for an application with 8 tasks– Fig. 14 (b), dispersion happens for an application with 12 tasks– Fig. 14 (c). This motives our next contribution to select the *first node* in a wiser manner.

[1] A node may have up to 4 available immediate neighbors.

ALGORITHM 1: The contiguous neighborhood allocation (CoNA) mapping algorithm.

```
input     : The application to be mapped: $A_p = TG(A, E)$.
            $t_f$: The *fist task* of $A_p$.
            $N'$: Set of available nodes.
output    : $map : T \to N'$
variables: $t_c$: the current task to be mapped; initialized to $t_f$.
            $t_p$: parent of $t_c$.
            $n_f$: The *first node* to be mapped on.
            $\bar{N} \subset N'$: Set of candidate nodes for mapping $t_c$.
1 $n_f \leftarrow$ The node $\in N'$ with the highest number of available neighbors and
    the least MD to system master;
2 map($t_c$) $\leftarrow n_f$ ;
3 repeat
4 |   $t_c \leftarrow$ next task in breadth-first order;
5 |   $\bar{N} \leftarrow$ nodes $\in N'$ with the least MD to map($t_p$);
6 |   map($t_c$) $\leftarrow$ the node $\in \bar{N}$ which fits into the smallest square with $n_f$;
7 until all tasks are mapped;
```

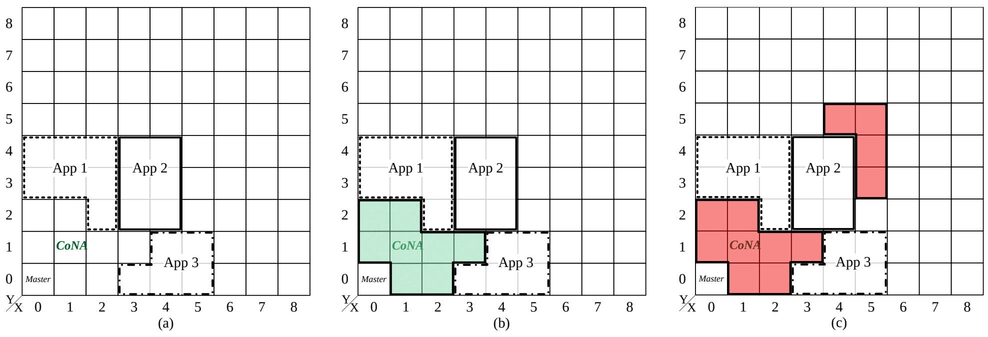


Figure 14. CoNA does not consider the application size in its *first node* selection method.

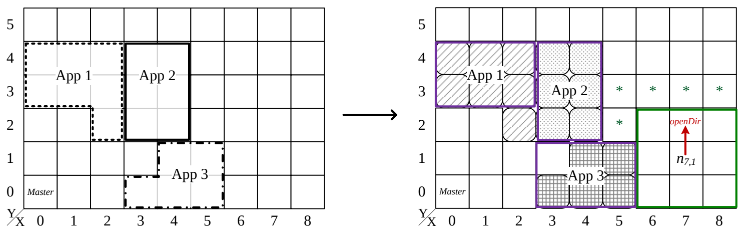


Figure 15. The square factor calculation for $n_{7,1}$.

To overcome this issue, we need to count the number of available nodes around a candidate *first node* , and compare it with the application size. The closer number to the application size, the better option for the *first node* selection. Accordingly, we approximate the number of available nodes around a candidate *first node* using our *square factor* metric. The $squarefactor($ SF$)_{x,y}$ approximates the number of contiguous available nodes around $n_{x,y} \in N'$ which are organized in an almost square shape. To calculate SF$_{x,y}$, we first model each running application as a rectangle, and then find the largest square centered around $n_{x,y}$, where it has no overlap with mesh borders nor existing rectangles (currently running applications). This is shown in Fig. 15 for $n_{7,1}$. The *square factor* value will be the square area plus the nodes around it, which are not in the rectangles (marked with asterisk); i.e. SF$_{7,1}$$= 9+5 = 14$. In other words, this node is the best *first node* for an application with 14 tasks.

Given the square factor of each node, our SHiC algorithm [39] tries to find a node with SF value close to the application size. To avoid exhaustive search over the mesh area for finding the node with the best SF value, we adapt random-restart stochastic hill climbing. SHiC is equipped with an $openDir$ value calculated along with SF, which helps the random walk to be wiser and reach to optimal nodes faster.

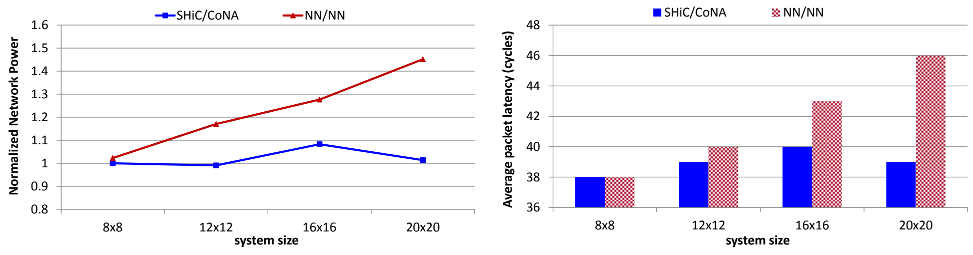
3.3.3 Communication Performance Evaluation

In this section, we evaluate the communication power and latency of our proposed *first node* selection and mapping algorithms. We modeled a 16x16 many-core processor which is 80% loaded, with applications of size 4 to 35 tasks. We tried different *first node* selection algorithms in combination with mapping algorithms. Table 2 shows the extracted average network latency (L_{avg}) and NoC power dissipation per flit (E_{flit}) normalized to SHiC case. As can be seen, our SHiC *first node* selection method used with CoNA mapping algorithm obtained over 50% reduction in network power dissipation compared to INC algorithm, while the network latency is decreased by 23%.

Results also demonstrate the high influence of *first node* selection method, compared to the mapping algorithm itself. When CoNA mapping algorithm is used with

Table 2. Communication power and latency results of different *first node* and run-time mapping methods, derived from [39].

mapping	CoNA					INC [24]	
first node	SHiC	Exh. SF	CoNA	NN [19]	INC	SHiC	INC
L_{avg}	40	39	41	43	42	42	52
E_{flit}	1.00	0.97	1.07	1.23	1.20	1.06	1.48



(a) Normalized NoC power **(b)** Average packet latency

Figure 16. Scalability of results quality over system size

INC *first node* selection, the power dissipation is increased by 29%, while the opposite combination (i.e. using SHiC with INC mapping) leads to only 4% more power dissipation.

3.3.4 Scalability Analysis

The time complexity of our CoNA mapping algorithm is $O(|T| \times |N|)$, which is linear to both network and application size. Also, the SHiC method works in polynomial time $O(|N|^2)$. This is faster than searching for contiguous regions $O(|N|^3)$, with worse results), and significantly better than deterministic clustering methods with NP-hard complexities. Furthermore, we studied how the quality of results scale with the system size, as shown in Fig. 16. Moving from a 36-core processor to a 400-core processor, when the processor is 80% loaded, the SHiC/CoNA method adds up to the NoC power by only 13% while the average packet latency is kept almost the same. On the other hand, the sate-of-the-art method, NN, results in linear increase of network power and latency with the system size.

3.4 Adjustable Contiguity

As results demonstrated in our previous contributions, improving contiguousness of allocations reduces the power dissipation and latency of the NoC fabric, which in turn, decreases the execution time of applications. Hence, limiting applications to map only onto contiguous node can significantly improve their execution time and the NoC power. As a contiguous allocation may not always exist, however, applications have to wait until the required contiguous region is freed. In turn, the actual turn-around time of applications will increase, which makes strictly-contiguous allocation a downgrading solution. This reasoning motivates the related work (including our previous contributions) to come up with techniques to improve the contiguousness of the mapping, while dispersion is not a restriction. When a contiguous allocation is not possible, further nodes are explored until the required number of PEs are collected. In other words, while a contiguous allocation is preferred, they tend to stay non-contiguous mapping algorithms.

In this contribution of the thesis, we look at the intermediate area between the two extreme cases of non-contiguous and strictly-contiguous mapping. That is, we do not restrict to only contiguous mappings, but put a limit on how much dispersion is allowed. We propose contiguity adjustable square allocation (CASqA) algorithm, where an α parameter (shown as superscript) is defined as the contiguity probe; i.e. CASqA$^{0.0}$ and CASqA$^{1.0}$ stand for the two ends of the spectrum– strictly-contiguous and non-contiguous, respectively.

At a high-level, CASqA works as follows. As explained in previous section, a mapping region is preferred to be square shaped. Accordingly, after allocating the *first node* (provided by SHiC), CASqA explores the first square around it (a 3x3 square) and looks for available nodes. Once all the available nodes in the current square are allocated, the square is expanded (into a 5x5 square) and nodes of next layer are explored. The square expansion continues until the square size (e.g. the 5x5 square has 25 nodes) becomes greater than or equal to the application size; i.e. $|T| \leq Area_{square}$. If all the tasks are already allocated at this point, this means that the application could be mapped on a contiguous squared-shaped region. Otherwise, the expansion continues conditionally, given an iterator counter $i = 1$. At each iteration, the expansion continues only if $\#\, unallocated\, tasks < |T| \times \alpha^i$. In CASqA$^{0.0}$ case, the square expansion stops at this point, while the stop condition is never satisfied for CASqA$^{1.0}$, and expansion continues until all tasks are mapped. As an example let us assume mapping of an application with 22 tasks with CASqA$^{0.5}$. At first, the exploration square is expanded up to 5x5, where 25 nodes are already explored. At this point, the square is expanded (to 7x7) if less than $22 \times 0.5 = 11$ tasks are remained unallocated. After all nodes of the 7x7 square are explored, we expand the square (to 9x9) only if less than $22 \times 0.5^2 = 5.5$ tasks are left for allocation. As can be seen, CASqA provides the user or the operating system with a probe to adjust the level of desired contiguousness. In case that an application cannot be mapped around the selected *first node* with the given α values, new first nodes are tried until the application is mapped; i.e. a better *first node* is found or a suitable region is freed.

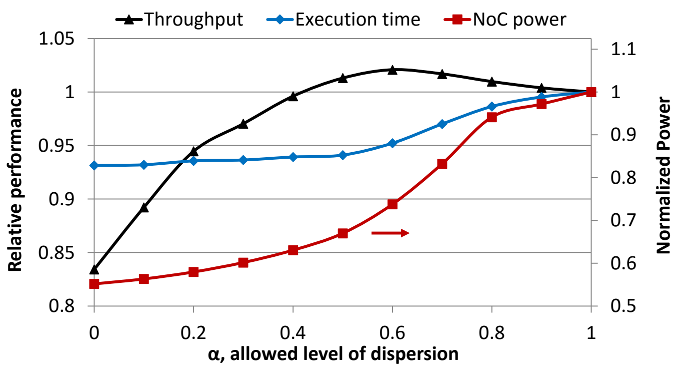


Figure 17. Normalized throughput of the system, along with average execution time and NoC power per application for different α values in CASqA method.

3.4.1 The Middle α Value

We evaluated the CASqA algorithm with different α values in 256-core processor on our system-C many-core simulator, running a set of 100 different synthetic applications with 4 to 35 different task. On average, tasks inject between 0.05 to 0.07 flits per cycle that is counted as a light load for the network.

Fig. 17 shows the normalized average execution time of applications, NoC power dissipated per application, and throughput of the whole system [2] as a function of the α value. As expected, while a strictly contiguous mapping improves the execution time of each application ($\tilde{8}\%$), it reduces the system throughput up to 15%. However, for α values around 0.6, the execution time improvements dominates the additional waiting time of applications, resulting in slightly increased throughput ($\tilde{3}\%$) compared to $\alpha = 1$ case.

Although the throughput improvements might seem negligible, it is obtained in addition to around 25% NoC power saving per application, compared to the non-contiguous case. From a different perspective, with $\alpha \approx 0.4$, the same throughput as the non-contiguous mapping can be obtained while the NoC power is reduced by 35%. In comparison, NN [19] method shows 56% and 35% higher NoC power and average packet latency, respectively, compared with CASqA($\alpha = 0.5$).

Optimal α value: The optimal α value will depend on the application. The more communication intensive an application is, the smaller optimal α value will get. In our experiments, communication-intensity of applications are similar. A more thorough analysis and further algorithm development are required to dynamically adapt the α value to the application characteristics.

[2] Throughput is defined as the number of applications executed per unit of time.

Table 3. Packet latency characteristics using CASqA$^{\alpha}$ mapping algorithm.

	$\alpha = 0.0$	$\alpha = 1.0$	*Relation*
Average	38.74	54.91	42%
Standard deviation	20.05	57.69	2.9 times
Worst-case	638	8663	13.6 times

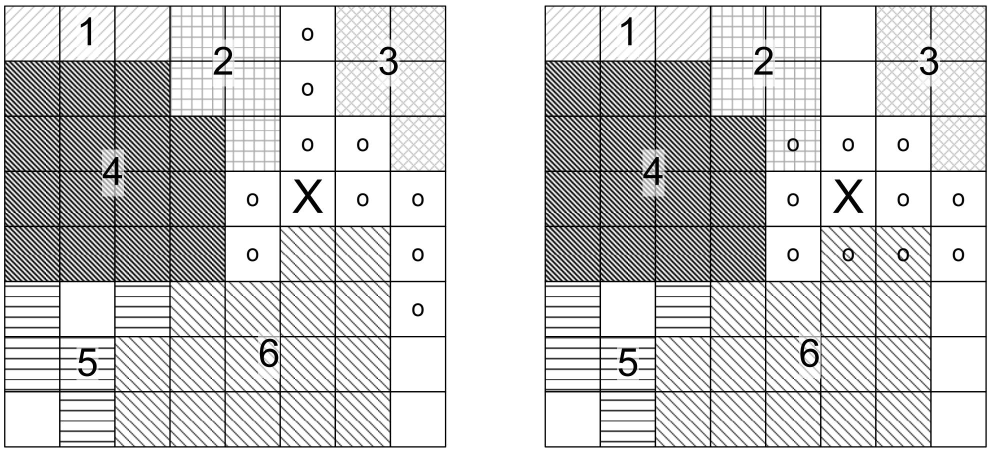
3.5 Providing QoS

Applications running on many-core processors have different characteristics and requirements. While some are average-performance-oriented, real-time applications have tight performance constraints. They have strait timing constrains [90], where tasks must meet completion time bounds (i.e. deadline) to correctly perform their computation. Communication deadlines are one of the main aspects that affect functionality of real-time applications. That is, the packet latency of the application traffic in the NoC infrastructure must be lower than a specific bound. In contrast with related works that propose architectural methods (e.g. priority virtual channels), as next contribution of this dissertation, we utilize our CASqA method and its insights to improve the quality of packets delivery for soft real-time applications.

In CASqA$^{0.0}$, the execution time of applications is reduced due to the reduction of the average packet latency. More importantly, our analysis reveal that contiguous application mapping significantly improves the worst-case and standard deviation of packet latencies, as summarized in Table 3. This motivates us to force contiguous mapping for real-time applications to meet more deadlines and improve the offered quality-of-service. As a result, once a real-time application is requested for execution, instead of expanding the exploration square, we locate real-time tasks on unavailable PEs of the same square. This is shown in Fig. 18, where the real-time application (shown by “`o`") is mapped contiguously (Fig. 18b) instead of being scattered around (Fig. 18a) the *first node* (indicated by “`X`"). Moreover , once a real-time task is co-located on a PE by other tasks, in our technique, the kernel scheduler prioritizes it over other tasks. In other words, the CPU is passed to other tasks only when the real-time task is suspended due to an undelivered `receive` call, and is preempted back as soon as it arrives.

3.5.1 Evaluation of Missed Deadlines

To examine effectiveness of our method, we added three real-time applications to our application set: MPEG-4, UAV, and VOPD. Accordingly, we measured the amount of missed deadline for each application, as shown in Fig. 19. We compared our method (QoS-mapping) to the case where our technique is disabled (norm.-mapping); i.e. all applications, including real-time ones, are mapped as usual. Moreover, to com-



(a) Normal application mapping. **(b)** Real-time application mapping.

Figure 18. Improving the QoS through contiguous application mapping.

pare with architectural methods, we simulated the case where the traffic of real-time applications is isolated from and prioritized over normal traffic in the network (QoS-NoC). Note that results are normalized to the `norm.-mapping` case. As can be seen, our proposed algorithm significantly improves the quality-of-service offered by real-time applications. In addition, our system-level method outperforms architectural enhancements for some applications, while their combination achieves the best results.

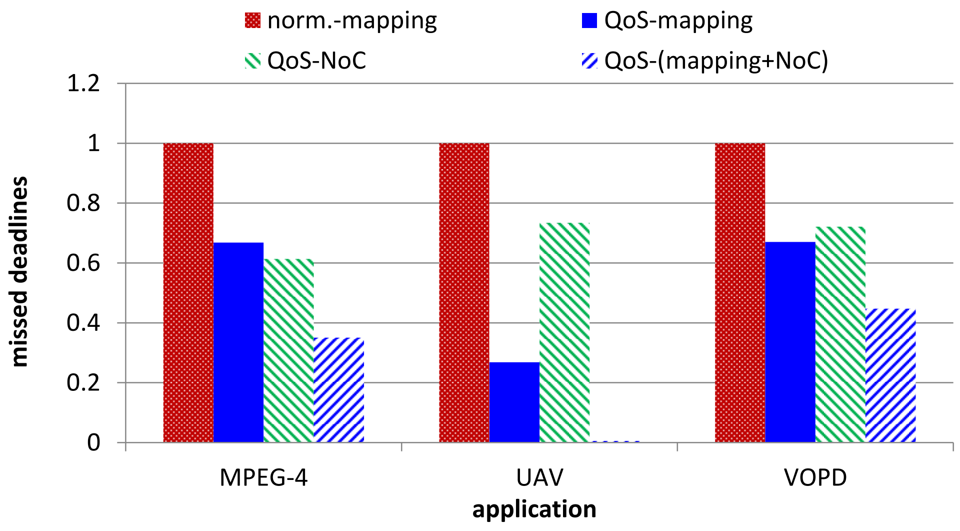


Figure 19. Missed deadlines of different real-time applications.

3.6 Test-Aware Mapping

Until now, contributions of our run-time application mapping methods are focused on power and performance of many-core architectures. In order to widen the scope of this thesis, in the following contributions, we take into account the decreasing reliability of on-chip components. Accordingly, in our first contribution, we consider the case where online testing methods are frequently executed on different cores. While online testing methods are necessary to ensure the continuous correct functioning of cores, there are two caveats associated with them. Though shortly, online testing keeps the cores under test busy where no task can be mapped on them, hindering the system throughput and performance. Moreover, test methods acquire a high power budget as they try to activate and examine all components in a short time.

The testing frequency needs to be proportional to the stress it has been affected with; i.e. the more intensive a core is being utilized, the more critical to be tested. Accordingly, each core is assigned a test criticality value ($tc_{x,y}$) associated , where $tc_{i,j} > 0.0$ indicates that the corresponding core needs to be tested. However, there might not be enough power budget available for testing at the moment or the core may be allocated, making it impractical to test the core immediately.

Accordingly, we prevent the cores with $tc_{i,j} > 0.0$ from being allocated to new applications, leaving them available for later testing. In order to prevent dispersion of applications due to untested cores, we exclude such cores from our SF calculations. Hence, an application is not dispersed due to a core being test critical. On the other hand, there might be several candidate cores to select from when possible to run test methods. Among the available cores with $tc_{i,j} > 0.0$, we select the ones that enhance mapping of future applications; i.e. the cores that lead to more contiguousness. Therefore, cores with higher SF values are prior for being tested. We define a test ranking $tr_{x,y}$ metric that integrates the $\mathrm{SF}_{x,y}$ value with the $tc_{x,y}$ value:

$$tr_{x,y} = tc_{x,y} + \frac{\sqrt{SF_{x,y}}}{|N|} \tag{8}$$

Once possible to put a core under test, the one with the highest tr value is selected. Note that the square root of the SF is calculated to limit its impact to the cases where two cores have close tc values.

3.6.1 Overhead Evaluation

In order to measure benefits of our proposed method, we simulated a system using three different methods. First, a system where there is no online testing. Second, our method, and third, a method where a fixed number of cores can go under test (fixed_test) with no specific algorithm for test scheduling [50].

Fig. 20 shows the extracted throughput of each method for different system sizes.

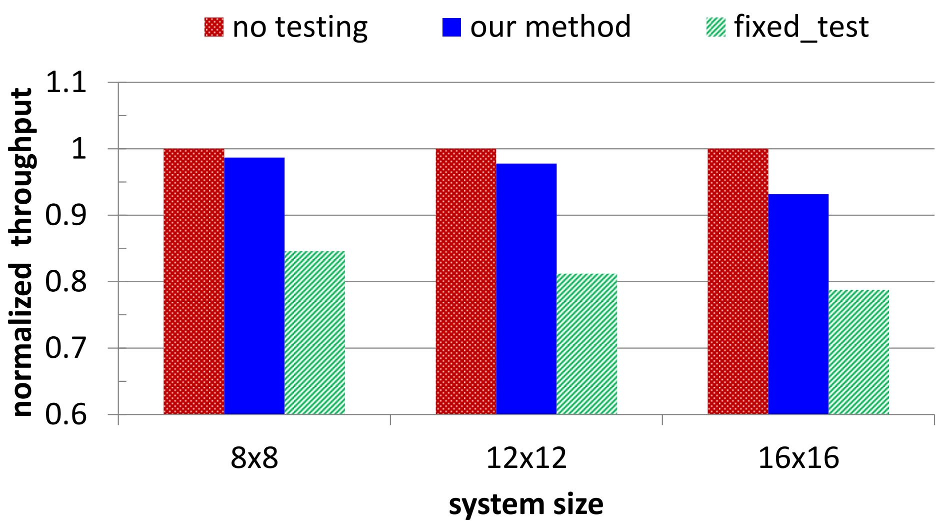


Figure 20. Normalized system throughput using different test scheduling methods.

As graphs demonstrate, we can obtain a high throughput level compared to naive approaches, very close to a system with no online testing overhead. Also, our method scales with the system size, with slight performance drop.

3.7 Fault-Aware Application Mapping

Online testing methods will pinpoint the components that become faulty. A faulty component may be part of the PE or the NoC components. When some link or routers fail in the network, some node pairs cannot exchange packets anymore, with the conventional NoC architectures. Several researchers have proposed architectural methods to overcome the situation. In contrast, we propose to turn around network failures at system-level; i.e. by exploiting application mapping. The motivation is to minimize the architectural overhead, while improving the fault-tolerance. To the best of our knowledge, this is the first work considering the imperfect network in its run-time application mapping algorithm for many-core architectures.

In this contribution, we propose system-level hierarchical fault-aware (SHiFA) approach to tolerating faults of NoC components. SHiFA is composed of two main parts: the hierarchy, and the fault-aware run-time mapping. That is, once a link or router fails, our method does not try to find a solution to keep the connectivity between disconnected nodes. Instead, it maps the applications such that their communication does not require the faulty component.

An important complication is that in order for a core to be allocated, it needs to be accessible by the system master. However, this can severely limit the utilizable

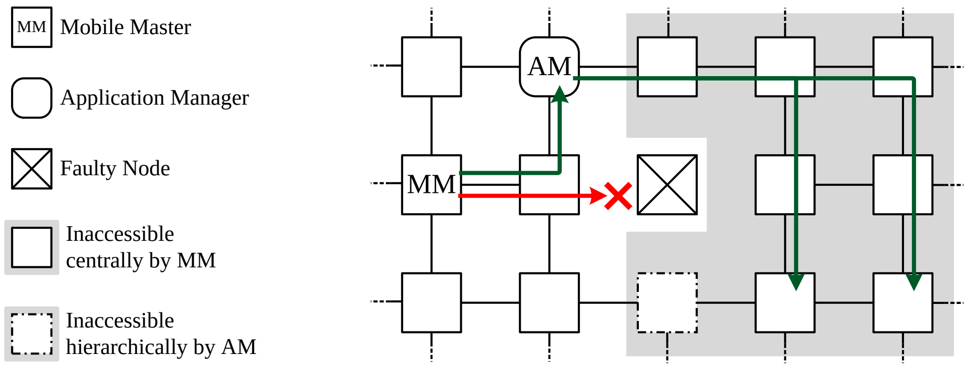

MM Mobile Master
Application Manager
Faulty Node
Inaccessible centrally by MM
Inaccessible hierarchically by AM
AM
MM

Figure 21. The system-level hierachy of SHiFA increases the utilizable nodes. The XY routing is assumed.

cores. In order to overcome the issue, SHiFA utilizes a system-level hierarchy. In other words, the kernels running on each core are grouped into 3 categories. First, the system master is equipped with a mobility feature and turns into a mobile master (MM), responsible for the whole system. Application managers (AMs) run in the second layer of the hierarchy, being responsible for mapping and (fault) management of their own applications. In the lowest level, normal kernels are, as usual, multitasking and handling the transport layer for the running tasks. As a result, in order for a core to be allocated, it does not necessarily need to be reachable by the MM but just the AM. This is illustrated in Fig. 21, where shaded nodes cannot be reached by the MM. Nonetheless, the selected AM can utilize 7 of them. Note that a kernel can play each of the roles, based on the run-time circumstances and SHiFA procedures, which will be explained below.

The mobile master: It coordinates the whole system and keeps the big picture of it, with two major responsibilities, AM selection and mobility. Upon entering new applications, it promotes a normal kernel to an AM. We utilize our SHiC algorithm incorporated with tabu search [47] to pick candidate nodes. If a kernel rejects the MM request for promotion, it is sent to the tabu list, until the current configuration of the system changes; e.g. when an application is terminated, or a new fault is occurred. An AM may reject the promotion request as it has been busy, or fails to find a feasible mapping for the application. On successful AM promotions, the AM reports the rectangle model (refer to Fig. 15) of its allocated cores back to MM. Accordingly, instead of the full system picture, MM only keeps the rectangular model of running applications. Mobility of MM is defined to increase number of reachable nodes and prevent it from being isolated. Upon new failures, MM may migrate to another core with a higher accessibility to other cores. As MM does not keep heavy information (as it is distributed among AMs), the migration is expected to be light.

Application managers: They map, migrate, and remap their application of responsibility. Once a kernel is assigned to map an application (promotion request), it starts exploring the cores surrounding it, and tried to find a feasible mapping for the application. It sends them reservation request, and considers them in allocation in case of positive response. In case a feasible mapping is found, the AM allocates tasks to the reserved cores and releases those cores not finally included in the mapping. Otherwise, it releases all cores it reserved during exploration. Finally it reports back to MM about the mapping result. To find a feasible mapping, AMs adapt a fault-aware variation of CASqA algorithm, where the exploration square can be extended until twice the application size cores are explored. This ensures a contiguous mapping, while limits the complexity of mapping. On the other hand, if a new failure that inflicts the application communication happens, AM tries to resolve the matter by migration few tasks or remapping the application from scratch. If no solution is found, the MM can be involved to migrate or restart the application in a healthier region.

3.7.1 Reliability of SHiFA

We implemented SHiFA in our System-C simulator to evaluate different aspects of it. We also added 9 different real applications to the application repository to have more realistic results, namely MPEG4 decoder [11], a video object plane decoder (VOPD) [100], and double video object plane decoder (DVOPD) [83], a generic multi-media system (MMS) which includes a H.263 video encoder, a H.263 video decoder, a MP3 audio encoder and a MP3 audio decoder [55], a picture-in-picture application (PIP) [56] and a multi-window display (MWD) application [100]. Unless for scalability analysis, we utilized a 12x12 processor in our simulations.

In the first reliability analysis, we study the enhancements achieved by proposed hierarchy. Accordingly, portion of nodes that can be promoted as AM along with those that can be allocated by AMs are measured. Results show that 1̃00% of nodes can potentially be allocated by AMs, when up to 8 links are broken in the network. We generated 100 random fault patterns, where we obtained full (100%) accessibility in all patterns. However, as our space exploration is not inclusive, we use the notion of 1̃00% for accuracy. Results also demonstrate that a more sophisticated routing algorithm enhances the obtained results, especially for higher fault rates, while it is not mandatory.

In addition for PEs to be accessible by AMs, feasible mappings require that the communication between tasks are not needing the failed components. Hence, the high rate of PEs accessible by AMs does not directly translate to feasible mappings.

As our next reliability measurement, we analyzed the average success rate of

Table 4. Success rates for arbitrary AM selection, derived from [41].

Method		SHiFA		D&C
NoC routing		XY	OE	OE
broken	1	99.85	99.93	99.52
links	8	98.04	99.41	96.47
	10%	89.19	96.45	85

AMs in mapping applications. Accordingly, for a given fault pattern, all potential AMs are assigned to map each of the 9 real applications, where the success rates are averaged per fault count. Table 4 shows the extracted results. For the sake of comparison, we also implemented a fault-aware adaptation of the distributed D&C mapping algorithm [6]. For instance, when up to 10% of the links are broken using the odd-even (OE) routing algorithm, an arbitrary AM can find a feasible mapping with 96.45% chance for the application. In other words, SHiFA may fail to map an incoming application 4.2 times less than D&C method. Note worthy, as SHiFA utilizes the CASqA mapping algorithm, its mapping quality in terms of NoC power and latency outperforms related work.

3.7.2 SHiFA Scalibility Evaluation

For the scalability analysis, we compared SHiFA with literature in two aspects. First, how many AMs requests are generated before a successful AM are found. Second, how much time is spent on executing the mapping algorithm. The scalability is analyzed over the number of cores and the fault count. In all settings, SHiFA finds a successful AM on average in 5 times less trials than random distributed methods (e.g. [6; 63; 53]). Besides having 50% faster computation time when there is no fault, SHiFA mapping algorithm experiences only 60% increase when up to 10% of the routers are failed, compared to D&C method with 16 times increase. Moreover, the mapping time increases by 80 times in D&C when moving form 36-cores to 256-cores, while again, SHiFA only shows 50% increase in computation time.

3.7.3 Limitations of SHiFA

As results demonstrated, SHiFA is able to map applications at run-time in a fault-aware manner. The proposed hierarchy greatly increased the allocatable PEs, while SHiC combined with tabu search significantly decreased the mapping trials compared to random scenarios. Finally, our fault-aware mapping algorithm, derived from CASqA, is able to find a feasible mapping in most of the times even in high fault rates.

Nonetheless, there are two assumptions in SHiFA, which limits its applicability

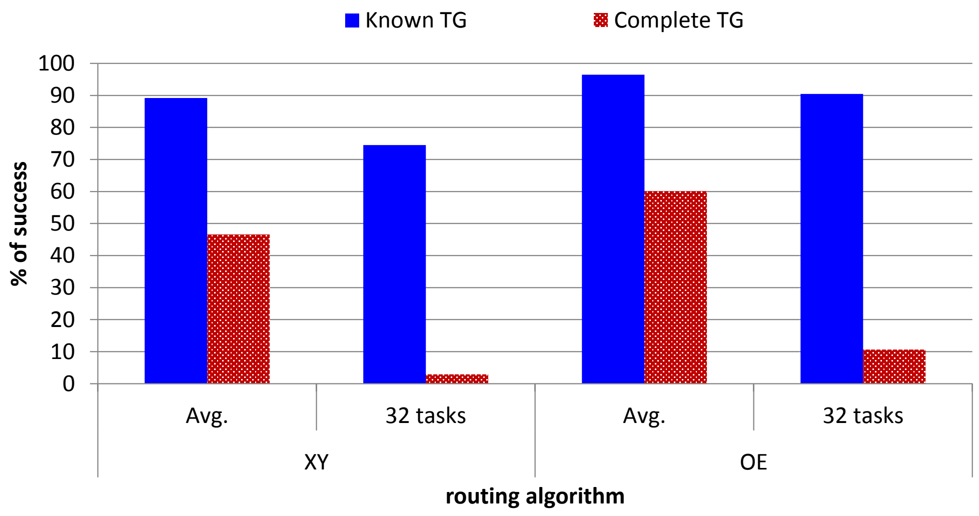


Figure 22. Comparing the success rate of arbitrary AM selection, when the application TG is known or assumed to be complete.

and potentials in practice. First, SHiFA, like most of fault-tolerant methods (e.g. [45; 80; 13; 87; 86; 92]), assumes a fault-free control network. That is, SHiFA requires a guaranteed negotiations between kernels. AMs need to report back to MM and normal kernels to their AM. In order to resolve this, we suggested, though in a separate virtual channel, to use the same faulty network for control packets, while violating the turn-model rules. This can potentially lead to deadlock for control packets, though the probability would be low.

Second, SHiFA relies on the assumption that the communication pattern of each application is known; i.e. its task graph. As this may not be always the case, SHiFA needs to assume a complete task graph for unknown applications; i.e. an application where all the tasks want to communicate with each other. Hence, the possibility to find a feasible mapping will significantly drop. This is shown in Fig. 22 for 144-core processor with 10% of faulty links, where applications are assumed to have the same number of tasks as our set of real applications. The possibility to find a feasible mapping has an adverse relation to the application size. For instance, when the task graph (TG) is to be unknown in DVOPD application with 32 tasks, there is only 10% (or even 3%) chance that an arbitrary AM can find a feasible mapping using odd-even (or XY) routing algorithm.

Note that other run-time mapping contributions of this dissertation make the same assumption of the TG being known. However, their correct functionality is not relied on this assumption. Particularly since we focus on contiguity of the mapped regions, where the within region allocation is a side work of our contributions.

This motives the next direction of contribution explore in this thesis. That is, the design of an architecture-level method to tolerant run-time permanent faults. A fault-tolerant routing algorithm, with no reliance on any fault-free components, with high fault coverage.

3.8 Summary

In this chapter, we briefed our run-time mapping algorithms towards communication-centric management of many-core systems. First, we demonstrated benefits of contiguous allocation, and proposed methods to improve contiguity of mapping results. Through which, we obtained significant network power and latency improvements, while stayed scalable for high core counts. Then, we took a deeper look at contiguous mapping, and proposed CASqA. Significant power saving achieved by limiting the allowed dispersion of the mapping, without hindering the system throughput. This led us to proving better quality-of-service, by limiting real-time applications to strictly contiguous mapping solutions. Results showed a meaningful degrade of missed deadlines.

Tackling the reliability challenge, we proposed two main solutions. A mechanism to run online-testing methods with minimized throughput penalties, and SHiFA, a fault-aware mapping algorithm that deals with faulty networks. While providing noticeable performance records, our methods proved to scale well with the core count and fault count.

In the next chapter, we explore our fault-tolerant routing algorithm that is motivated by the limitations of our SHiFA mechanism.

4 Fault-Tolerant Routing

As the last direction of our contributions, we cope with the reliability issues of future many-core processors at architecture level. While failures may happen all over the chip, we focus on the faults affecting the NoC routers and links, since a healthy communication substrate is critical to correct functioning of the whole system. Accordingly, we propose a fault-tolerant routing algorithm, called maze-routing.

We set four goals in our design aiming at making our algorithm practical. First, due to the extremely reduced reliability of aggressively shrunk transistors, we cannot be limited to the fault count. Our algorithm needs to tolerate any number of failure regardless of their location in the network, and moreover, detect disconnected nodes. We name this goal **full (fault) coverage**. Second, as also explained in previously in Chapter 3.7.3, we must not assume any oracle fault-free component in the network. Hence, a practical routing algorithm needs to be **fully-distributed**, where each component works independently. Third, a practical algorithm should not lead to heavy implementations overheads, as each single transistor contributes to the failure rate of a component. Thus, it needs to have **low area overhead**. Fourth, as explained in Chapter 3.6, components going under test are disabled similar to faulty ones; though for a short period but frequently. Disabling network components should not interrupt the network operation, as they are tested periodically and frequently. As a result, in order to be practical, a fault-tolerant method needs to impose **low reconfiguration overhead**. Elaborated in Chapter 4.1, to the best of our knowledge, maze-routing is the first to satisfy all of these goals together.

4.1 Related Works

There are plenty of methods to detect and tolerate different types of faults (permanent, transient, etc.) in NoC components. To be concise, however, we only explore works similar to our Maze-routing algorithm, in which a *routing algorithms* is proposed to tolerate *permanent faults* in NoC links and routers. As we set four goals in the design of a *practical* fault-tolerant routing algorithm[1], we explain each work based on its relation to these goals, summarized in Table 5.

Zhang et al. [107] prose a variation of X-Y routing algorithm which reconfig-

[1] A detailed explanation of each goals is presented in Chapter 4.2.

Table 5. Comparison of state-of-the-art. Desirable characteristics are in bold [36].

	Coverage	*Reconfiguration*	*O(Area)*	*O(Reconf.)*
Zhang et al. [107]	few	**fully dist.**	**low**	**on the fly**
LBDR [87]	moderate	central	**low**	N/A
d^2-LBDR [13]	moderate	central	**low**	N/A
OSR-Lite [92]	moderate	central	high	moderate
TOSR [9]	moderate	distributed	high	**fast**
BLINC [66]	moderate	distributed	high	**fast**
Wachter et al. [101]	high	distributed	high	slow
Fick et al. [45]	high	distributed	high	slow
Face routing [17]	high	**fully dist.**	excessive	**on the fly**
FTDR-H [43]	high	**fully dist.**	high	**fast**
uLBDR [86]	**full**	central	high	excessive
uDIREC [80]	**full**	central	high	excessive
ARIADNE [2]	**full**	distributed	high	slow

ures if one of its 8 surrounding routers are faulty. The algorithm imposes low area overhead, works in a distributed manner, and reconfigures according to the fault pattern on the fly. However, the algorithm can only work with one-faulty-router (or one-faulty-region) cases.

In order to implement adaptive routing algorithms without routing tables, a set of algorithms are proposed based on logic-based distributed routing (LBDR) method [13; 87; 86]. Initially, Rodrigo et al. [87] introduce LBDR method and an extension of it— LBDR-extended (LBDRe). LBDR can be reconfigured, using 12 bits per router, to implement a variety of different routing algorithms with a very simple logic. LBDRe require 24 configuration bits, and can support a wider range of routing configurations. Hence, LBDR and LBDRe impose very low area overhead while covering a moderate number of NoC faults. Nevertheless, these methods need a centralized controller to calculate reconfiguration bits of each router, based on the fault pattern. Authors later introduce universal-LBDR (uLBDR) to achieve full coverage, which uses 18 configuration bits per router. However, in order for the method to work "an *exhaustive search* algorithm that tests all the paths in a recursive way for every source-destination pair" is needed. Moreover, as reported in [13], uLBDR imposes 3x area overhead as it needs virtual cut through switching with FORK modules and complex arbitration. To remove the high area overhead of uLBDR, authors propose d^2-LBDR method [13]. It uses a distance register and a new deroute mechanism, which removes the switching and arbitration complexity of uLBDR. Accordingly, d^2-LBDR removes the area overhead of uLBDR with sacrificing the achieved fault coverage.

In their OSR-Lite method, Strano et al. [92] utilize two instances of uLBDR to achieve faster reconfiguration and avoid deadlock while transitioning from one routing configuration to another one. Similar to uLBDR, OSR-Lite imposes high area overhead and requires a central controller, while provides a better reconfiguration speed (few hundreds of clock cycles). Later, Balboni et al. [9] try to improve over OSR-Lite, by proposing tunneled OSR (TOSR) method. They take advantage of multiple physical NoCs, available at some many-core systems (e.g. Tile microprocessors [106]). As such, TOSR works in a distributed manner and reconfigures the network upon new faults much faster than OSR-Lite (within tens of clock cycles). Nevertheless, it provides a moderate fault coverage while imposes high area overheads– it uses uLBDR in each physical router along with a small routing table.

One of the first distributed algorithms dealing with high number of failures is proposed by Fick et al. [45]. Upon new faults, their algorithm works in lockstep and reconfiguration messages are exchanged between reconfiguration units residing in each router. However, in their paper they assume that “routers know when they need to invoke the algorithm and how to resume operation after reconfiguration finishes". This introduces a synchronization point in the algorithm and breaks the distributed manner of the algorithm. ARIADNE [2] is introduced later to mitigate this problem and provide full fault coverage. Similarly, routers of ARIADNE work in lockstep and exchange reconfiguration messages. One router (the one detecting the new failure) initiates the reconfiguration and other routers operate normally, until the reconfiguration message is propagated to them. On the other hand, these works pause the normal operation of the network for a long time. For instance, it takes 10K cycles for ARIADNE to reconfigure the network and revert its normal operation. Authors of the same research group, later propose BLINC [66] to overcome the slow reconfiguration of ARIADNE, sacrificing the full fault coverage. It reconfigures the network within tens of cycles and works in a distributed manner. All of these works, however, use routing tables and impose high area overheads.

In a different work, Parikh et al. [80] introduce uDIREC. Unlike other works that disable both direction links in case of failure, they use a finer fault model where a link can be faulty in one direction and healthy in other direction. Accordingly, more number of links are available to the network to utilize, and some PE may get isolated later than usual. Their work, however, uses a centralized controller with excessive reconfiguration complexity; e.g. it takes around 100 ms to reconfigure the network. In addition, they take use of routing tables that imposes high area overhead.

While one of the main challenges of the explained works is to avoid deadlock in case of failures, deflection-based routers [75] have the interesting feature of being deadlock-free in essence. Similar to our Maze-routing algorithm, Feng et al. [43] take advantage of this property in their proposed FTDR-H algorithm. They use reinforcement learning algorithms to learn optimal paths to destinations and update the routing tables accordingly. FTDR-H works in a fully distributed manner and converges to the optimal path quickly, whereas, its routing table imposes a high area overhead.

As an inspiration to our Maze-routing algorithm, face routing algorithm [17] is developed for ad hoc wireless networks. Face routing has interesting features that satisfy most two main goals of our contribution. It works in a fully distributed manner and finds the path to destination on the fly; i.e. there is no reconfiguration phase and overhead. It also guarantees to find a path to destination. However, it does not provide a method to detect it when destinations become unreachable. Moreover, it needs to store real (vs. integer) values in the packets header and requires floating-point operations at each input port of the router. While, floating-point operations are known for their area and power hungry characteristics.

In summary, as can be seen in Table 5, none of the previously presented works satisfy all design goals of this dissertation at the same time. They do not provide full fault coverage, or if do so, impose high area and reconfiguration overheads. They might work in a fully distributed manner, but then other goals are left unaddressed. Whereas our Maze-routing algorithm achieves all at once.

4.2 Maze-Routing Algorithm

Maze-routing stores 4 variables in the packet header: MD_{best}, $mode$, N_{trav} and DIR_{trav}. The first to are used to deliver the packet to destination, while other two are to detect when the destination is not reachable. The distributed implementation of maze-routing algorithm is presented in IX. However, for the sake of simplicity, Algorithm 2 describes its functionality from a central point of view. A packet may be routed either in $normal$ (line 4) or $traversal$ (lines 6-19) modes. Initially, a packet is in $normal$ mode and each router tries to forward it to a productive output; i.e an output in which the MD to destination is decreased. In case a productive output does not exist, the packet enters the $traversal$ mode, wherein right/left hand-rules (↱ / ↰) are selected arbitrarily. Accordingly, the selected hand-rule is followed for selecting the output to forward the packet to. The $traversal$ mode continues until it is safe to exit to $normal$ mode (line 12), or the destination is detected unreachable (line 14). When a path between source and destination exists, maze-routing algorithms guarantees that we will exist $traversal$ mode (to the $normal$ mode) in a finite step. Since a packet in $normal$ mode gets one hop closer to destination at each iteration (by taking a productive output), this guarantees that we will definitely reach destination in a finite number of steps. On the other hand, the destination is detected unreachable (line 13), if (i) the packet returns to its initial router in which entered the $traversal$ mode–$cur = \text{N}_{trav}$ and (ii) is about to repeat the same path–$\text{DIR}_{trav} = \text{DIR}_{hand}$. This implies that the packet will turn around the network through the same path *forever* without being delivered to destination. Maze-routing algorithm guarantees that the packet is detected unreachable *iff* there is no path to destination. A more thorough explanation of the algorithm along with the formal proof of its correct functionality is presented in IX.

ALGORITHM 2: The maze-routing algorithm.

```
input     : $src$: Coordinates of the source router.
            $dst$: Coordinates of the destination router.
output    : Path from $src$ to $dst$ or indicating an unreachable $dst$.
variables: $cur$: Coordinates of the current router.
            $\textsc{MD}_{best}$: The smallest MD to $dst$ we ever reached so far.
            $hand - rule$: The hand-rule followed in the $traversal$ mode.
            $\textsc{DIR}_{hand}$: The direction in which the $hand - rule$ directs to.
            $\textsc{N}_{trav}$: Coordinates in which we enter $traversal$ mode.
            $\textsc{DIR}_{trav}$: Direction at which we exit $\textsc{N}_{trav}$ for first time.

$\textsc{MD}_{best} \leftarrow \textsc{MD}(src, dst)$;
while $cur \neq dst$ do
    if $\exists$ a productive output then
        Take the productive output;
    else
        $\textsc{MD}_{best} \leftarrow \textsc{MD}(cur, dst)$;
        $\textsc{N}_{trav} \leftarrow cur$;
        $hand\ rule \leftarrow \Rsh\ /\ \Lsh$;
        Imagine a line between $cur$ and $dst$;
        $\textsc{DIR}_{trav} \leftarrow$ the first output in the left/right of the line;
        Take $\textsc{DIR}_{trav}$ and go to the next router;
        while $\textsc{MD}(cur, dst) \neq \textsc{MD}_{best}$ or $\nexists$ a productive output do
            if $cur = \textsc{N}_{trav}$ and $\textsc{DIR}_{trav} = \textsc{DIR}_{hand}$ then
                return $dst$ is unreachable;
            else
                Follow $\textsc{DIR}_{hand}$;
            end
        end
    end
end
return $dst$ reached;
```

As can be intuitively seen, maze-routing algorithm needs to have no limit on the output directions it picks. In a wormhole switched network, this can lead to deadlock. To avoid deadlocks, we take advantage of deflection based router architectures [75; 33] where deadlock is prevented in essence with possibility of arbitrary output selection. However, flits might get deflected to the outputs other than their request, making deflected packet not to follow maze-routing algorithm anymore. To resolve this and keep the guarantees of maze-routing, we reset the algorithm every

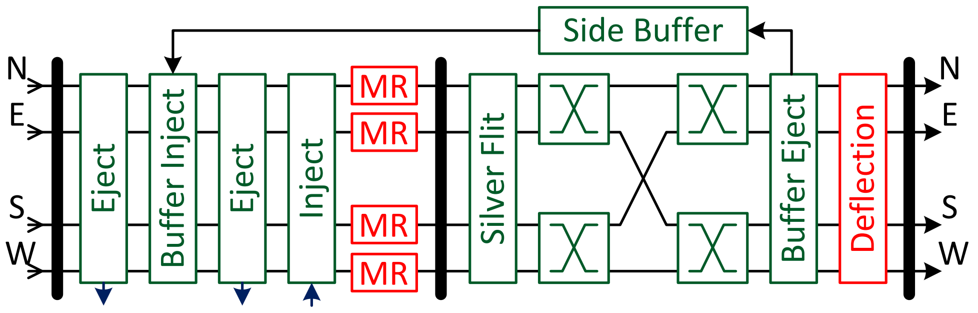


Figure 23. The maze-routing algorithm implemented in the minBD [33] architecture.

time the flit is deflected. That is, once a flit is deflected to a router other than what maze-routing directs to, the algorithm is restarted from line 1; i.e. it exits to $normal$ mode and/or resets the MD_{best} value accordingly.

4.3 Experimental Setup

We used NOCulator [32; 33] to implement and validate our maze-routing algorithm. The main motivation for using NOCulator is its support of deflection based router architectures as needed in our algorithm, whereas, Noxim simulator only supports wormhole switching strategy. Similar to Noxim, NOCulator supports a wide range of configuration parameters. Unlike HeMPS and our in-house simulator, on the other hand, NOCulator models shared-memory many-core architectures. Accordingly, we validated Maze-routing using shared-memory SPEC2006 benchmarks [52]. We released the source code of the developed algorithm and the router architecture on GitHub [37] as a branch of the original simulator.

4.4 Results

As explained in Chapter 1.5, we use NOCulator [32] to model an 8x8 processor with deflection-based routers. We implemented our maze-routing algorithm inside the minBD router [33], and published the source-code on GitHub [37]. The resultant router architecture is shown in Fig. 23. The deflection module is in charge of resetting the MD_{best} and $mode$ values if the flit is deflected.

Reliability: As mentioned, we formally prove that maze-routing algorithm will find the path to destination, and detects when no path exists. This is achieved while each router only knows about the health status of its own links. Hence, the fault-tolerance is delivered in a fully-distributed manner. As there is one instance of maze-routing algorithm per input port (shown as `MR` in Fig. 23), transistor failures in each module only disables the associated input/output link, while the rest of the router can work normally. In addition, we measured the area overhead of our routing algorithm in comparison to the smallest routing table utilized in literature that provides full fault coverage, like us. Maze-routing implementation requires 3.8 and 15.9 times less silicon area for 8x8 and 16x16 meshes, respectively.

Performance: We ran two sets of experiments to measure performance of maze-routing algorithm. First, a set of simulations where 1 link in the network is disabled and put under test is executed. Results demonstrate instantaneous adaptation of the network with only 0.25 cycle increase in the average network latency per disabled link, when nodes are injecting to the network with a rate as high as 0.2 flits/cycle. In comparison, the network operation is interrupted for at least tens of cycles [9; 66] for works with limited coverage, while it takes 40,000 cycles for the network to retain its steady-state latency once a link is disabled in methods with full fault coverage [2]. Finally, we compared the average network latency and saturation throughput of an 8x8-processor under synthetic and real application traffics, when up to 5 random links are broken. Since maze-routing is fully-distributed, its path selection may become suboptimal and lead to non-minimal routes. Accordingly, in low injection rates, it shows around 0.5 to 1.0 cycles higher average network latency than state-of-the-art for both synthetic and real traffics. On the other hand, thanks to the offered path divergence, it leads to 50% higher saturation throughput under synthetic traffic and up to halved average latency for real applications.

5 Description of Papers

5.1 Overview of Original Papers

5.1.1 Paper I- Exploration of MPSoC Monitoring and Management Systems

As MPSoCs are rapidly growing with their core count, their resource monitoring and management is becoming more challenging than ever. In this paper, we review different monitoring and management schemes used in many-core MPSoCs. In addition, we make a qualitative comparison between distributed and central schemes. To have the scalability of distributed methods, while keeping the holisticness of a central one, hierarchical management systems are advocated.

Author's contribution: The author contributed by reviewing the state-of-the-art, write-up and presentation, under the guidance of coauthors. Mohammad Fattah is the main author of this paper.

5.1.2 Paper II- Transport Layer Aware Design of Network Interface in Many-Core Systems

In parallel processing platforms, including the many-core SoCs, data communication is one of the major bottlenecks. A significant effort is done in literature to reduce the network latency. In a message-passing architecture, the OS is providing the transport-layer services to the communicating tasks. In this paper, we demonstrate that the OS overhead in providing transport services has a much bigger impact on the communication performance, compared to underlying network. As a solution, we propose a NI architecture (Tra-NI) with understandings about the transport layer, in which, the OS software overhead is offloaded to the NI hardware. Simulation results demonstrate 4 times improvement in the end to end network latency and 4.7 times higher achievable communication throughput.

Author's contribution: The main idea presented in this paper, its implementation, write-up and presentation was developed by the author under the guidance of the coauthors. Mohammad Fattah is the main author of this paper.

5.1.3 Paper III- CoNA: Dynamic Application Mapping for Congestion Reduction in Many-Core Systems

In this paper, we propose an application mapping algorithm for dynamic workload of many-core systems. Our contiguous neighborhood allocation (CoNA) algorithm tries to minimize on-chip network congestion at two levels. Inter application congestion by aiming a contiguous set of nodes for a given application, and intra application congestion by placing the communicating tasks in a close proximity. Our algorithm is composed of three main novelties. 1) First node selection; to improve the contiguity of selected nodes, CoNA takes the status of neighboring nodes into account when selecting the first node. 2) First task to map; to provide the largest possible number of available nodes around the first task, our algorithm selects the task with the largest number of edges to be mapped onto the first node. 3) Contiguous neighborhood allocation; to minimize inter-application congestion, among suitable nodes for a task, CoNA selects the one which fits in the smallest square with the first node. Our fully synthesizable design shows 16% less average network latency, and up to 40% improvement in different network metrics.

Author's contribution: The main idea presented in this paper, SW implementation, write-up and presentation was developed by the author, while the FPGA implementation and bring-up was realized by Marco Ramirez, under the guidance of the coauthors. Mohammad Fattah is the main author of this paper.

5.1.4 Paper IV- Smart Hill Climbing for Agile Dynamic Mapping in Many-Core Systems

In this paper, we utilize stochastic hill climbing algorithm to improve our first node selection method. We first motivate the importance of first node selection in many-core systems, where not only a contiguous set of nodes is desired, but also it is desired to incur minimum fragmentation to the remaining available nodes. Then, we approximate the available area around a given node, namely the square factor. Finally, our smart hill climbing (SHiC) algorithm climbs the network nodes to find the one that has the required number of nodes around it, using the square factor. The application mapping starts around the selected first node. Followed by our CoNA mapping algorithm, SHiC provides the best network performance compared to existing heuristics in the literature. Moreover, our results show that other heuristics would benefit significantly from using SHiC as their first node selection method, which proves the importance of the first node selection algorithm.

Author's contribution: The main idea presented in this paper, implementation, write-up and presentation was developed by the author under the guidance of the coauthors. Also, the author developed the simulation environment described in Section 3.2 of the thesis, and Section 5 of this paper, on top of the Noxim simulator. Mohammad Fattah is the main author of this paper.

5.1.5 Paper V- Adjustable Contiguity of Run-Time Task Allocation in Networked Many-Core Systems

In this paper, we propose a run-time mapping algorithm where the dispersion level of allocated nodes (α) can be controlled. In other words, the OS or the user can select to allocate an application to strictly contiguous nodes ($\alpha = 0.0$), to any set of nodes with best effort contiguousness ($\alpha = 1.0$), or something in between. Strictly contiguous allocation improves the network performance and the execution time of applications. However, as applications need to wait until a contiguous allocation is possible, it limits the achievable throughput of the system and increases the turnaround time of applications. As a result, in order to obtained the maximum system throughput, state-of-the-art considers only best effort contiguousness, with no dispersion limit on the allocated nodes. On the contrary, our results demonstrate that even a higher system throughput (by 3%) is possible when some dispersion limit is imposed ($\alpha < 1.0$). More importantly, by limiting the dispersion level of allocated nodes, one can achieve the same system throughput (compared to $\alpha = 1.0$), while reducing the network power dissipation by 35%.

Author's contribution: The main idea presented in this paper, implementation, write-up and presentation was developed by the author under the guidance of the coauthors. Mohammad Fattah is the main author of this paper.

5.1.6 Paper VI- Mixed-Criticality Run-Time Task Mapping for NoC-Based Many-Core Systems

In this paper, we propose to take advantages of strictly contiguous task allocation methods for mapping of applications with soft real-time requirements. Contiguous allocation reduces network congestion between different applications. As a result, it considerably improves the network latency, its worst-case and the standard deviation. On the other hand, for non-critical applications, we do no strict to only contiguous mapping to improve the system throughput. Our results demonstrate up to 50% better worst-case latencies and even 100 time less deadline misses for some applications.

Author's contribution: The main idea presented in this paper, implementation and presentation was developed by the author, under the guidance of his supervisory team. The paper was also written jointly by authors. Mohammad Fattah is the main author of this paper.

5.1.7 Paper VII- A Power-Aware Approach for Online Test Scheduling in Many-core Architectures

In this paper, we propose an online test scheduling algorithm for power limited many-core systems. We first define a test critically metric to determine the cores that need to go under test. However, as test routines are power hungry in nature, there might not be enough power budget for a core to be tested immediately, leading to scenarios where there are more cores to be tested than the power limit permits. A test ranking metric is defined that incorporates the run-time status of allocated nodes with their test criticallity, to increase the priority of the cores that benefit the system performance as well. Our results show significant improvement in system throughput (~15%) compared to methods that have no specific algorithm for test scheduling.

Author's contribution: The author proposed the power aware mapping algorithm and contributed to the write-up of the associated section of the paper.

5.1.8 Paper VIII- SHiFA: System-Level Hierarchy in Run-Time Fault-Aware Management of Many-Core Systems

In this paper, we introduce a hierarchical resource management scheme for the run-time application mapping of many-core systems. The scheme takes into account the faultiness of the network nodes and links, and ensures that the communication of an application tasks are feasible given the routing algorithm and the partial network. Upon arrival of a new application, we we first find an accessible node to offload the application mapping to it. The node then searches its neighboring accessible nodes for finding a feasible mapping. In case the node fails to find a feasible mapping, due to existing faults, a new node is searched to offload the mapping. Using simply XY routing in a 12x12 network and when up to 8 links are broken, our method shows up to 100% PEs being utilizable, and around 98% of mapping requests being successful.

Author's contribution: The main idea presented in this paper, implementation, write-up and presentation was developed by the author under the guidance of the coauthors. Mohammad Fattah is the main author of this paper.

5.1.9 Paper IX- A Low-Overhead, Fully-Distributed, Guaranteed-Delivery Routing Algorithm for Faulty Network-on-Chips

In this paper, we propose the maze-routing algorithm for the faulty NoCs. Maze-routing is the first-in-class algorithm that provides all of the followings together: it is fully distributed, guarantees the delivery, imposes low area overhead, and adapts to new faults on the fly. The algorithm provides 16x less area overhead compared to other algorithms that provide guaranteed delivery, while can achieve up to 50% higher saturation throughput.

Author's contribution: The author contributed to the development of the main idea, the presentation and most of the simulations and write-up, under the guidance of the supervisory team. Mohammad Fattah is the main author of this paper.

6 Conclusion and Discussion

The number of cores on many-core SoCs continues to grow as we have reached the limits of a single core performance. In this dissertation, we demonstrated the importance of the communication overhead in the design of many-core SoCs, and put forward several contributions to tackle these challenges, in three main directions.

First, we demonstrated the significance of the end points overhead in the communication performance, and proposed a network interface that would offload the work from application and kernel SW instances (Paper II). After radically squeezing the contribution of the network interface, we started to see the importance of run-time application mapping in the communication and overall system performance. This led us to the second direction of this dissertation, where we developed several methods for run-time application mapping. We showcased the importance of contiguous application mapping in improving network performance (Paper III and Paper IV). Then, we elaborated that contiguousness can be exploited, counter-intuitively, to improve the throughput of the system (Paper V), and to provide quality-of-service in a system with soft real-time applications (Paper VI). Later, we studied how run-time application mapping can help the testability and fault-tolerance of many-core systems. We reduced the impact of putting cores under test on the system throughput (Paper VII), in the emergence of dark-silicon many-core SoCs. We also proposed SHiFA, a hierarchical mapping approach to tackle the faultiness of network elements (Paper VIII). SHiFA, however, suffered from the assumption of a fault-free control network, through which, the fault information of the main NoC of the system is distributed between the nodes. Tackling this shortcoming paved our path to the third direction of this dissertation. We proposed the maze-routing algorithm (Paper IX), the first algorithm that could provide all of the claimed advantages at the same time.

Throughout this thesis, wherever applicable, we assumed a message-passing model for both the applications and the system, in contrast to a shared-memory paradigm. In the early stages of this work, there was a live debate on how the future of many-core SoCs will look like. Some favoring shared-memory systems for its obvious SW benefits, while others were speculating the rise of the message-passing paradigm due to the practical limits of building coherent many-core SoCs. Intel SCC platform was one example of early SoCs built, supporting the latter. However, such SoCs are not commercially available these days, showing the importance of SW demands (i.e., the industry goes where the customer wants). Moreover, in our system model, we assumed that each PE had its own local memory accessible to it, without a need to go off the node (to a far cache or off-chip DRAM). This assumption is motivated by the technology advancements allowing 3D integration of DRAM dies on top of logic silicons.

6.1 Future Work

This dissertation can be advanced in several directions, to explore new ideas and to improve some of the shortcomings. In this section, we go through some of the thoughts.

Multitasking: In the mapping algorithms proposed in this dissertation, we assumed cores running a single task at a time. However, cores are usually multitasking in commercial environments. To advance the depth of this research, adding the multitasking angle to the run-time mapping algorithms can be pursued. Finding optimum collocations brings significant challenges to the algorithm design.

Mixed contiguity: In our CASqA approach, we assumed the same α value for all the running applications. However, applications are different in their network insensitivity. The more the tasks of an application would need network access, the more it benefits from contiguous mapping. One possible future direction is to study the optimum α value according to the application profile, and how different applications would impact each other.

Migration: In this thesis, we assumed the mapping is fixed, once it is decided. However, the system and even the individual applications may benefit from migrating few tasks to more contiguous nodes once a free space is available. This especially seems advantageous when multitasking is part of the system. We believe there is a lot to discover in this direction.

Shared memory: In our opinion, this research work can be enriched significantly by considering the shared-memory SoCs. We think that similar to message-passing applications, one may see benefits in using traits of our innovations in shared-memory systems. For instance, by limiting the network traffic and conflict induced by an application with assigning a limited number of contiguous last level cache (LLC) slices to it, according to the application and overall system circumstances.

Path optimization: While maze-routing guaranteed the delivery in presence of faulty network elements, it does not guarantee to find an optimum path. The algorithm is deciding on local information which leads to suboptimal paths, and deflection routing adds to the number of hops that a flit may traverse under heavy traffic. Last but not least, we suggest that improving the path selection in maze-routing can be studied as a future direction. Methods like ant-colony can be used to keep the algorithm fully-distributed.

List of References

[1] The Story of the Intel 4004. `http://www.intel.com/museum/archives/4004.htm`. [Online; accessed 27-December-2015].

[2] Konstantinos Aisopos, Andrew DeOrio, Li-Shiuan Peh, and Valeria Bertacco. ARIADNE: Agnostic Reconfiguration In A Disconnected Network Environment. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 298–309. IEEE, 2011.

[3] George S Almasi and Allan Gottlieb. *Highly Parallel Computing*. Menlo Park, CA (USA); Benjamin-Cummings Pub. Co., 1988.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] Abdel Krim Amoura, Evripidis Bampis, and Jean-Claude Konig. Scheduling Algorithms For Parallel Gaussian Elimination with Communication Costs. *Parallel and Distributed Systems, IEEE Transactions on*, 9(7):679–686, 1998.

[6] Iraklis Anagnostopoulos, Alexandros Bartzas, Georgios Kathareios, and Dimitrios Soudris. A Divide and Conquer based Distributed Run-time Mapping Methodology for Many-Core platforms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 111–116. IEEE, 2012.

[7] Marjan Asadinia, Mehdi Modarressi, Arash Tavakkol, and Hamid Sarbazi-Azad. Supporting Non-contiguous Processor Allocation in Mesh-based CMPs Using Virtual Point-to-point Links. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[8] Brahim Attia, Wissem Chouchene, Abdelkader Zitouni, and Rached Tourki. Network Interface Sharing for SoCs based NoC. In *Communications, Computing and Control Applications (CCCA), 2011 International Conference on*, pages 1–6. IEEE, 2011.

[9] Marco Balboni, José Flich, and Davide Bertozzi. Synergistic Use of Multiple On-Chip Networks for Ultra-Low Latency and Scalable Distributed Routing Reconfiguration. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 806–811. EDA Consortium, 2015.

[10] Carl M Bender, Michael A Bender, Erik D Demaine, and Sándor P Fekete. What is the optimal shape of a city? *Journal of Physics A: Mathematical and General*, 37(1):147, 2004.

[11] Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, Rutuparna Tamhankar, Stergios Stergiou, Luca Benini, and Giovanni De Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):113–129, February 2005.

[12] Raoul AF Bhoedjang. *Communication Architectures for Parallel-Programming Systems*. Amsterdam: Vrije Universiteit, 2000.

[13] Rimpy Bishnoi, Vijay Laxmi, Manoj Singh Gaur, and José Flich. d^2-LBDR: Distance-Driven Routing to Handle Permanent Failures in 2D Mesh NoCs. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 800–805. EDA Consortium, 2015.

[14] S. Borkar. Design Challenges of Technology Scaling. *Micro, IEEE*, 19(4):23–29, Jul 1999.

[15] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *Micro, IEEE*, 25(6):10–16, 2005.

[16] Shekhar Borkar. Thousand Core Chips: A Technology Perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[17] Prosenjit Bose, Pat Morin, Ivan Stojmenović, and Jorge Urrutia. Routing with Guaranteed Delivery in Ad Hoc Wireless Networks. *Wireless networks*, 7(6):609–616, 2001.

[18] E.A. Carara, R.P. de Oliveira, N.L.V. Calazans, and F.G. Moraes. HeMPS - A Framework for NoC-based MPSoC Generation. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 1345–1348, May 2009.

[19] Ewerson Carvalho, Ney Calazans, and Fernando 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. IEEE, 2007.

[20] Ewerson Carvalho, Ney Calazans, and Fernando Moraes. Dynamic Task Mapping for MPSoCs. *Design & Test of Computers, IEEE*, 27(5):26–35, 2010.

[21] Vincenzo Catania, Andrea Mineo, Salvatore Monteleone, and Maurizio and Palesi. Noxim: An Open, Extensible and Cycle-accurate Network on Chip Simulator. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 162–163. IEEE, 2015.

[22] Chen-Ling Chou and Radu Marculescu. Run-Time Task Allocation Considering User Behavior in Embedded Multiprocessor Networks-on-Chip. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(1):78–91, 2010.

[23] Chen-Ling Chou and Radu Marculescu. FARM: Fault-Aware Resource Management in NoC-Based Multiprocessor Platforms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[24] Chen-Ling Chou, Umit Y Ogras, and Radu 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, 2008.

[25] Wissem Chouchene, Brahim Attia, Abdelkrim Zitouni, Nouredine Abid, and Rached Tourki. A low power network interface for network on chip. In *Systems, Signals and Devices (SSD), 2011 8th International Multi-Conference on*, pages 1–6. IEEE, 2011.

[26] William J Dally and Brian Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689. IEEE, 2001.

[27] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Elsevier, 2004.

[28] Masoud Daneshtalab, Masoumeh Ebrahimi, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. Memory-Efficient On-Chip Network with Adaptive Interfaces. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(1):146–159, 2012.

[29] David P Dobkin, Herbert Edelsbrunner, and Mark H Overmars. Searching for Empty Convex Polygons. *Algorithmica*, 5(1-4):561–571, 1990.

[30] Thomas Ebi, David Kramer, Wolfgang Karl, and Jörg Henkel. Economic Learning for Thermal-aware Power Budgeting in Many-core Architectures. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on*, pages 189–196. IEEE, 2011.

[31] Jr John Presper Eckert and John W Mauchly. Electronic numerical integrator and computer, February 4 1964. US Patent 3,120,606.

[32] Chris Fallin and Rachata Ausavarungnirun. NOCulator. `https://github.com/CMU-SAFARI/NOCulator/`. [Online; accessed 27-December-2015].

[33] Chris Fallin, Greg Nazario, Xiangyao Yu, Kuo-Pin Chang, Rachata Ausavarungnirun, and Onur Mutlu. MinBD: Minimally-Buffered Deflection Routing for Energy-Efficient Interconnect. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2012.

[34] Al Faruque, Mohammad Abdullah, Rudolf Krist, and Jörg Henkel. ADAM: Run-time Agent-based Distributed Application Mapping for on-chip Communication. In *Proceedings of the 45th annual Design Automation Conference*, pages 760–765. ACM, 2008.

[35] Mohamamd Fattah, Marco Ramirez, Masoud Daneshtalab, Pasi Liljeberg, and Juha Plosila. CoNA: Dynamic Application Mapping for Congestion Reduction in Many-Core Systems. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 364–370. IEEE, 2012.

[36] Mohammad Fattah, Antti Airola, Rachata Ausavarungnirun, Nima Mirzaei, Pasi Liljeberg, Juha Plosila, Siamak Mohammadi, Tapio Pahikkala, Onur Mutlu, and Hannu Tenhunen. A Low-Overhead, Fully-Distributed, Guaranteed-Delivery Routing Algorithm for Faulty Network-on-Chips. In *Proceedings of the 9th International Symposium on Networks-on-Chip*, NOCS '15, pages 18:1–18:8, New York, NY, USA, 2015. ACM.

[37] Mohammad Fattah and Rachata Ausavarungnirun. The Maze-routing algroithm. `https://github.com/CMU-SAFARI/NOCulator/tree/Maze-routing/`. [Online; accessed 27-December-2015].

[38] Mohammad Fattah, Masoud Daneshtalab, Pasi Liljeberg, and Juha Plosila. Transport Layer Aware Design of Network Interface in Many-Core Systems. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–7. IEEE, 2012.

[39] Mohammad Fattah, Masoud Daneshtalab, Pasi Liljeberg, and Juha Plosila. Smart Hill Climbing for Agile Dynamic Mapping in Many-Core Systems. In *Proceedings of the 50th Annual Design Automation Conference*, page 39. ACM, 2013.

[40] Mohammad Fattah, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. Adjustable Contiguity of Run-Time Task Allocation in Networked Many-Core Systems. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 349–354. IEEE, 2014.

[41] Mohammad Fattah, Maurizio Palesi, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. SHiFA: System-Level Hierarchy in Run-Time Fault-Aware Management of Many-Core Systems. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6. IEEE, 2014.

[42] Mohammad Fattah, Amir-Mohammad Rahmani, Thomas Canhao Xu, Anil Kanduri, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. Mixed-Criticality Run-Time Task Mapping for NoC-Based Many-Core Systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 458–465. IEEE, 2014.

[43] Chaochao Feng, Zhonghai Lu, Axel Jantsch, Minxuan Zhang, and Zuocheng Xing. Addressing Transient and Permanent Faults in NoC With Efficient Fault-Tolerant Deflection Router. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(6):1053–1066, 2013.

[44] Alberto Ferrante, Simone Medardoni, and Davide Bertozzi. Network Interface Sharing Techniques for Area Optimized NoC Architectures. In *Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on*, pages 10–17. IEEE, 2008.

[45] David Fick, Andrew DeOrio, Gregory Chen, Valeria Bertacco, Dennis Sylvester, and David Blaauw. A Highly Resilient Routing Algorithm for Fault-Tolerant NoCs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 21–26. European Design and Automation Association, 2009.

[46] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[47] Fred Glover, Manuel Laguna, E Taillard, and D De Werra. *Tabu Search.* Baltzer Basel, 1993.

[48] Dennis Gnad, Muhammad Shafique, Florian Kriebel, Semeen Rehman, Duo Sun, and Jorg Henkel. Hayat: Harnessing Dark Silicon and Variability for Aging Deceleration and Balancing. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[49] Mohammad-Hashem Haghbayan, Amir-Mohamad Rahmani, Antonio Miele, Mohammad Fattah, Juha Plosila, Pasi Liljeberg, and Hannu Tenhunen. A Power-Aware Approach for Online Test Scheduling in Many-core Architectures. *Computers, IEEE Transactions on*, PP(99):730–743, 2015.

[50] Mohammad-Hashem Haghbayan, Amir-Mohammad Rahmani, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. Energy-Efficient Concurrent Testing Approach for Many-Core Systems in the Dark Silicon Age. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014 IEEE International Symposium on*, pages 270–275. IEEE, 2014.

[51] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[52] John L Henning. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[53] Mohammad Hosseinabady and Jose Luis Nunez-Yanez. Run-time stochastic task mapping on a large scale network-on-chip with dynamically reconfigurable tiles. *IET computers & digital techniques*, 6(1):1–11, 2012.

[54] John Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, Devon Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010.

[55] Jingcao Hu and Radu Marculescu. Energy- and performance-aware mapping for regular NoC architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(4):551–562, April 2005.

[56] Egbert G. T. Jaspers and Peter H. N. de With. Chip-set for video display of multimedia information. *IEEE Transactions on Consumer Electronics*, 45(3):706–715, August 1999.

[57] Solid State Technology Association JEDEC. Failure Mechanisms and Models for Semiconductor Devices. *JEDEC Publication JEP122G*, 2010.

[58] Anil Kanduri, Mohammad-Hashem Haghbayan, Amir-Mohammad Rahmani, Pasi Liljeberg, Axel Jantsch, and Hannu Tenhunen. . In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pages 573–580. IEEE, 2015.

[59] Nishit Kapadia and Sudeep Pasricha. VARSHA: Variation and Reliability-Aware Application Scheduling with Adaptive Parallelism in the Dark-Silicon Era. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1060–1065. EDA Consortium, 2015.

[60] Nishit Kapadia, Venkata Yaswanth Raparti, and Sudeep Pasricha. ARTEMIS: An Aging-Aware Runtime Application Mapping Framework for 3D NoC-based Chip Multiprocessors. In *Proceedings of the 9th International Symposium on Networks-on-Chip*, page 31. ACM, 2015.

[61] Heba Khdr, Santiago Pagani, Muhammad Shafique, and Jörg Henkel. Thermal Constrained Resource Management for Mixed ILP-TLP Workloads in Dark Silicon Chips. In *Proceedings of the 52nd Annual Design Automation Conference*, page 179. ACM, 2015.

[62] Daewook Kim, Manho Kim, and Gerald E Sobelman. NIUGAP: Low Latency Network Interface Architecture with Gray Code for Networks-on-Chip. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4–pp. IEEE, 2006.

[63] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. DistRM: Distributed Resource Management for On-Chip Many-Core Systems. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 119–128. ACM, 2011.

[64] Israel Koren and C Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann, 2010.

[65] Yong-Long Lai, Shyue-Wen Yang, Ming-Hwa Sheu, Yin-Tsung Hwang, Hui-Yu Tang, and Pin-Zhang Huang. A High-Speed Network Interface Design for Packet-Based NoC. In *Communications, Circuits and Systems Proceedings, 2006 International Conference on*, volume 4, pages 2667–2671. IEEE, 2006.

[66] Doowon Lee, Ritesh Parikh, and Valeria Bertacco. Brisk and Limited-Impact NoC Routing Reconfiguration. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

[67] Seung Eun Lee, Jun Ho Bahn, Yoon Seok Yang, and Nader Bagherzadeh. A Generic Network Interface Architecture for a Networked Processor Array (NePA). In *Architecture of Computing Systems–ARCS 2008*, pages 247–260. Springer, 2008.

[68] Daniele Ludovici, Alessandro Strano, and Davide Bertozzi. Architecture Design Principles for the Integration of Synchronization Interfaces into Network-on-Chip Switches. In *Network on Chip Architectures, 2009. NoCArc 2009. 2nd International Workshop on*, pages 31–36. IEEE, 2009.

[69] Milo MK Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, 2012.

[70] Timothy G Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, et al. The 48-core SCC processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[71] Mehdi Modarressi, Arash Tavakkol, and Hamid Sarbazi-Azad. Virtual Point-to-Point Connections for NoCs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(6):855–868, 2010.

[72] Gordon E Moore et al. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, 1965.

[73] Fernando Moraes. HeMPS Generator Framework. `https://corfu.pucrs.br/redmine/projects/hemps`. [Online; accessed 27-December-2015].

[74] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. HERMES: an infrastructure for low area overhead packet-switching networks on chip. *INTEGRATION, the VLSI journal*, 38(1):69–93, 2004.

[75] Thomas Moscibroda and Onur Mutlu. A Case for Bufferless Routing in On-Chip Networks. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 196–207. ACM, 2009.

[76] Wu Ning, Ge Fen, and Wu Fei. Design of a GALS Wrapper for Network on Chip. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, volume 3, pages 592–595. IEEE, 2009.

[77] Vincent Nollet, Prabhat Avasare, Hendrik Eeckhaut, Diederik Verkest, and Henk Corporaal. Run-Time Management of a MPSoC Containing FPGA Fabric Tiles. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):24–33, 2008.

[78] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-chip Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 2–11, New York, NY, USA, 1996. ACM.

[79] Tarik Ono and Mark Greenstreet. A Modular Synchronizing FIFO for NoCs. In *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 224–233. IEEE Computer Society, 2009.

[80] Ritesh Parikh and Valeria Bertacco. uDIREC: unified diagnosis and reconfiguration for frugal bypass of NoC faults. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 148–159. ACM, 2013.

[81] Davide Patti and Maurizio Palesi. Noxim: Network-on-chip simulator. `https://github.com/davidepatti/noxim`. [Online; accessed 27-December-2015].

[82] Fernando CN Pereira and Touradj Ebrahimi. *The MPEG-4 book.* Prentice Hall Professional, 2002.

[83] Antonio Pullini, Federico Angiolini, Paolo Meloni, David Atienza, Srinivasan Murali, Luigi Raffo, Giovanni De Micheli, and Luca Benini. Noc design and implementation in 65nm technology. In *International Symposium on Networks-on-Chip*, pages 273–282, 2007.

[84] Andrei Radulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees 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, 2005.

[85] Steve Rhoads. Plasma RISC microprocessor. `http://opencores.com/project,plasma`. [Online; accessed 27-December-2015].

[86] Samuel Rodrigo, Jose Flich, Antoni Roca, Simone Medardoni, Davide Bertozzi, Jorge Camacho, Federico Silla, and Jose Duato. Cost-Efficient On-Chip Routing Implementations for CMP and MPSoC Systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):534–547, 2011.

[87] Samuel Rodrigo, Simone Medardoni, Jose Flich, Davide Bertozzi, and Jose Duato. Efficient implementation of distributed routing algorithms for NoCs. *IET computers & digital techniques*, 3(5):460–475, 2009.

[88] Marshall T Rose. *The open book: a practical perspective on OSI*. Prentice-Hall, Inc., 1990.

[89] Muhammad Shafique, Dennis Gnad, Siddharth Garg, and Jörg Henkel. Variability-Aware Dark Silicon Management in On-Chip Many-Core Systems. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 387–392. EDA Consortium, 2015.

[90] Zheng Shi and Alan Burns. Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 161–170. IEEE Computer Society, 2008.

[91] Jens Sparsø, Evangelia Kasapaki, and Martin Schoeberl. An Area-efficient Network Interface for a TDM-based Network-on-Chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1044–1047. EDA Consortium, 2013.

[92] Alessandro Strano, Davide Bertozzi, Francisco Trivino, José L Sánchez, Francisco J Alfaro, and José Flich. OSR-Lite: Fast and Deadlock-Free NoC Reconfiguration Framework. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 86–95. IEEE, 2012.

[93] Alvin W Strong, Ernest Y Wu, Rolf-Peter Vollertsen, Jordi Sune, Giuseppe La Rosa, Timothy D Sullivan, and Stewart E Rauch III. *Reliability Wearout Mechanisms in Advanced CMOS Technologies*, volume 12. John Wiley & Sons, 2009.

[94] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[95] Andrew S Tanenbaum. *Computer networks, 5th edition*. Prentice Hall PTR, 2011.

[96] Andrew S Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, 2014.

[97] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE micro*, 22(2):25–35, 2002.

[98] Yvain Thonnart, Edith Beigné, and Pascal Vivet. Design and Implementation of a GALS Adapter for ANoC based Architectures. In *Asynchronous Circuits and Systems, 2009. ASYNC'09. 15th IEEE Symposium on*, pages 13–22. IEEE, 2009.

[99] Peggy Salz Trautman. A Computer Pioneer Rediscovered, 50 Years On. *The New York Times*, April 1994.

[100] Erik B. van der Tol and Egbert G.T. Jaspers. Mapping of MPEG-4 decoding on a flexible architecture platform. *Media Processors*, 4674:362–375, 2002.

[101] Eduardo Wachter, Augusto Erichsen, Alexandre Amory, and Fernando Moraes. Topology-Agnostic Fault-Tolerant NoC Routing Method. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1595–1600. EDA Consortium, 2013.

[102] David W Walker and Jack J Dongarra. MPI: a Standard Message Passing Interface. *Supercomputer*, 12:56–68, 1996.

[103] Hangsheng Wang, Li-Shiuan Peh, and Sharad Malik. Power-driven Design of Router Microarchitectures in On-chip Networks. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 105–, Washington, DC, USA, 2003. IEEE Computer Society.

[104] HangSheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: A Power-performance Simulator for Interconnection Networks. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 294–305, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[105] Andreas Weichslgartner, Stefan Wildermann, and Jürgen Teich. Dynamic Decentralized Mapping of Tree-Structured Applications on NoC Architectures. In *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*, pages 201–208. IEEE, 2011.

[106] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE micro*, (5):15–31, 2007.

[107] Zhen Zhang, Alain Greiner, and Sami Taktak. A Reconfigurable Routing Algorithm for a Fault-Tolerant 2D-Mesh Network-on-Chip. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 441–446. IEEE, 2008.

[108] Konrad Zuse. *The Computer— My Life*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

TURUN
YLIOPISTO
UNIVERSITY
OF TURKU

Painosalama, Turku, Finland 2021

ISBN 978-951-29-8708-5 (PRINT)
ISBN 978-951-29-8709-2 (PDF)
ISSN 2736-9390 (PRINT)
ISSN 2736-9684 (ONLINE)