Mapping of Processing Elements of Hardware-based Production Systems on Networks on Chip

by

Mostafa Wasiuddin Numan

B. Sc. (Engineering) in Computer Science and Engineering
Shahjalal University of Science and Technology, Bangladesh, 2006

M. Sc. (By Research) in Electrical, Electronic and Systems Engineering
National University of Malaysia (UKM), Malaysia, 2010

Thesis submitted for the degree of

Doctor of Philosophy

in

Electrical and Electronic Engineering
Faculty of Engineering, Computer and Mathematical Sciences
The University of Adelaide, Australia

2017

Supervisors:

Professor Michael Liebelt

Dr Braden J. Phillips

To my dearest parents,
to my wonderful wife, Fathima, and our little princess, Zoharin,
to my sister, Mili, and brother, Shahan,
without whom none of my success would be possible.

Contents

Declaration

	September 26, 2017
Signed	**Date**

Acknowledgments

First and foremost, I take this opportunity to convey sincere gratitude to my supervisors, **Professor Michael Liebelt** and **Dr Braden J. Phillips** for their guidance and support throughout my candidature. Their remarkable expertise and unwavering optimism have always been helpful in propelling my research forward. I owe many thanks to my principal supervisor, Professor Liebelt for his generous knowledge sharing, insightful discussions and inspiring words in times of needs. His critical and thoughtful comments were always constructive and fruitful to improve the quality of my research. I am thankful to Dr Phillips for being a constant source of research ideas and constructive suggestions. I would like to gratefully acknowledge his enthusiastic supervision and encouraging attitude.

I recognise that this research would not have been possible without the financial support of Australian Government via a generous International Postgraduate Research Scholarship (IPRS) and Australian Postgraduate Award (APA).

There are many people who helped me throughout my PhD candidature. I would express appreciation to my colleagues and friends in the Centre for High Performance Integrated Technologies and Systems (CHiPTec), Jesse Frost, Francis Li, Peng Wang, Thanh Thi Thanh Bui, Simon Keen and Muhammad Usman Khan for their valuable comments time to time. My sincere gratitude goes to Jesse for his passion in discussing around different research issues and critical suggestions.

I am thankful to the office and support staff of the school for their kindness and assistance. I would also like to thank the head of the school, Associate Prof. Cheng-Chew Lim, for his support during my PhD research. In addition, I sincerely thank my friends and colleagues in the school, Syed Imranul Islam, Mehdi Kasaei, Dr Amir Ebrahimi, Dr Sarah Immanuel, Robert Moric, Madhulika Tripathi, Yansong (Garrison) Gao, Nicholas P. Lawrence, Dr Sam Darvishi, Shengjian (Jammy) Chen, Dr Ali Karami, Dr Zahra Shaterian, Mariam Ebrahimpour, Dr Mostafa Rahimi and Nazmul Huda for making such a friendly research environment. It has been a privilege to get acquainted with so many talented individuals.

I appreciate the selfless support of my friends, Dr Md. Ayub, Kingshuk Nandy, Dr Manabendra Saha, Dr Raihan Rumman, Shuvra Shaha, Asafuddoulah Suzon, Javed Hussain, Hassan Jahangir and Dr Zaheed Hasssan for making life so easy since my very early days in Adelaide. I also thank my good old friends, Majba Uddin, Saiful Islam, Rumon Rahman, Zaidul Alam, Razeeb Saleheen, Amit Roy and their families for always staying around me and my family. You all make me feel at home while away from home.

My endless gratitude goes to my parents, who always bestowed me with infinite support, wishes and continuous love. They always wanted me to be a 'doctor' (physician). I somehow failed to pursue their dreams, but they never complained, rather encouraged me for whatever I did. I hope, they can now get a contentment seeing 'Dr' in front of my name. Abbu and Ammu, you are the best blessings of Almighty Allah to me. I also thank Apamoni, Dulabhai and Najnin for their tremendous support and inspiration throughout my journey. As always, my heartfelt gratitude goes to my younger brother, Shahan, for taking my responsibilities on his shoulder since very young age. I also wish to express my warm and sincere thanks to my mother-in-law for her kindness and earnest wishes.

Last but not least, my wholehearted appreciation are due to my beloved wife, Fathima. The way she took care of me and our little princess, Zoharin, besides her own study and work, was really commendable. My study would not have been completed so smoothly without her patience and sacrifices. She stood by me in ups and downs, not only during my PhD study, but also in all stages of my life, and always endowed me with her endless love and support. My dear, you are very special.

Mostafa Wasiuddin Numan
September 2017
Adelaide

Conventions

The following conventions have been adopted in this thesis:

Typesetting

This document was compiled using LaTeX2e. Texmaker and TeXstudio were used as text editor interfaces to LaTeX2e. Inkscape and Matlab were used to produce schematic diagrams and other drawings.

Referencing

The Harvard style has been adopted for referencing.

System of units

The units comply with the international system of units recommended in an Australian Standard: AS ISO 1000–1998 (Standards Australia Committee ME/71, Quantities, Units and Conversions 1998).

Spelling

Australian English spelling conventions have been used, as defined in the Macquarie English Dictionary (A. Delbridge (Ed.), Macquarie Library, North Ryde, NSW, Australia, 2001).

Abstract

This thesis investigates network on chip (NoC) architecture, most particularly, NoC mapping algorithms for homogeneous processing elements of a system on chip (SoC) designed for AI and cognitive computing.

Production systems are used in cognitive architectures and knowledge-based systems to produce appropriate reasoning behaviours by matching the symbolic information of the environment with the production rules stored in their knowledge bases. General purpose computers are not specifically manufactured for the purpose of continuous matching involved in production systems, and often fail to deliver the performance and speed required in real-time applications. A reconfigurable and parallel computer architecture, named the Street Processor, has been developed by the research group of which the author is a member, to address the performance gap. The processor has its own instruction set, called the Street language, to define the production rules. The production rules are implemented on simple and identical PEs of the Street Processor that conduct the matching operations in parallel and asynchronously. Special steps can be taken to make these operations synchronous if required. Two artificial agents demonstrate the capability of the Street Processor, and are also used as test cases to measure the performance of NoC mapping techniques.

The Street Processor is expected to contain thousands of fine-grained homogeneous PEs to build a complex cognitive agent. To make the continuous and simultaneous communication among the PEs more efficient, a regular and generic NoC architecture is considered in this work. The network architecture allows multiple PEs to be associated with a single NoC router to optimise its resources. The mapping of PEs to NoC routers, which is an NP-hard optimisation problem, is addressed in this work using two alternative approaches. The Branch and Bound (BB) and Simulated Annealing (SA) techniques are analysed for use as a preferred mapping technique. Although the BB technique provides a mapping solution faster than SA, the latter is considered more promising for large systems, e.g. the Street Processor, since BB achieves the computation time advantage at the cost of a high memory requirement.

To reduce the computation time of the SA method by shrinking the search space, the dependency graph, which captures the communication volume among PEs over a period of time, is partitioned into smaller groups of PEs (GPEs). By assigning each GPE to a router, this approach also reduces the number of required routers and the inter-router traffic of the network. A Priority-based Simulated Annealing (PSA) technique is proposed, which takes advantages of the relative placements of the routers and inter-dependencies of the GPEs to determine a heuristic initial mapping to start annealing. The experiments show that this approach significantly improves the computation time for finding a solution without sacrificing mapping quality. Considering the inherent memory utilisation advantage over the BB technique, and the computation time improvement over the original SA technique, the proposed approach is suggested to be the most suitable for NoC mapping for the Street Processor and similar homogeneous SoCs.

List of Figures

List of Tables

Chapter 1

Introduction

THIS thesis considers on-chip communication networks for parallel computation using a production system programming model. The context and motivation for the research are explained in this chapter. It briefly describes the significance of networks on chip (NoCs) as a communication platform among hardware processing elements (PEs). The original contributions of the research are highlighted, and an overview of the thesis structure is presented in this chapter.

1.1 Background

Since Allen Newell, Herbert Simon and Cliff Shaw wrote the first artificial intelligence (AI) program in 1956 to mimic the problem solving skills of human beings, research activity aimed at achieving human-like intelligence has progressed along many streams. This includes knowledge-based systems (Hayes-Roth and Jacobstein 1994), genetic and evolutionary algorithms (Goldberg 1994, Jones 1998), fuzzy systems (Zadeh 1994), neural networks (Haykin 2001) and artificial general intelligence (AGI) (Goertzel and Wang 2007). While most of the approaches aim at a particular aspect or application of intelligence, AGI targets the implementation of an intelligence that can be applied in various domains. Cognitive architectures have been proposed as the most elegant way to achieve AGI in artificial systems (Langley 2006), and have been successfully applied to many applications (Anderson 1983, Laird *et al.* 1987, Anderson 1996). They capture knowledge and skills in systems to yield intelligent behaviours in a diversity of complex environments. Cognitive architectures commonly use production systems for knowledge representation and processing. In production systems, the knowledge is encoded in the form of production rules to achieve appropriate reasoning behaviour in accordance with the states of the environment in which the systems are running. Besides cognitive architectures, production systems are also applied in the implementations of different knowledge-based systems in areas including computer aided design (Chen *et al.* 2012), medical diagnosis (Shortliffe 2012) and adaptive support systems (Angelov 2013), just to name a few.

In the years since that first AI program, semiconductor fabrication technology has advanced to the point that it is possible to cost-effectively produce integrated circuits (ICs) containing billions of transistors. For three decades the exponential increase of transistor count followed Moore's law, and it is only in recent years that the rate of growth has started to slow as shown in Figure 1.1 (Scherer 2015). The performance demands of modern complex embedded applications have also increased substantially. These demands cannot be handled by the processors containing a single core or processing element. The technology growth and increasing performance demands have driven researchers to adopt a new paradigm, called the system on chip (SoC), in which an entire computational systems is realised on a single chip. For very large and complex systems, SoCs often contain multiple PEs, and are then known as multi-processor SoCs (MPSoCs) (Jerraya and Wolf 2004). The underlying concept of such systems is to divide applications into sections that can be processed concurrently on multiple programmable PEs.

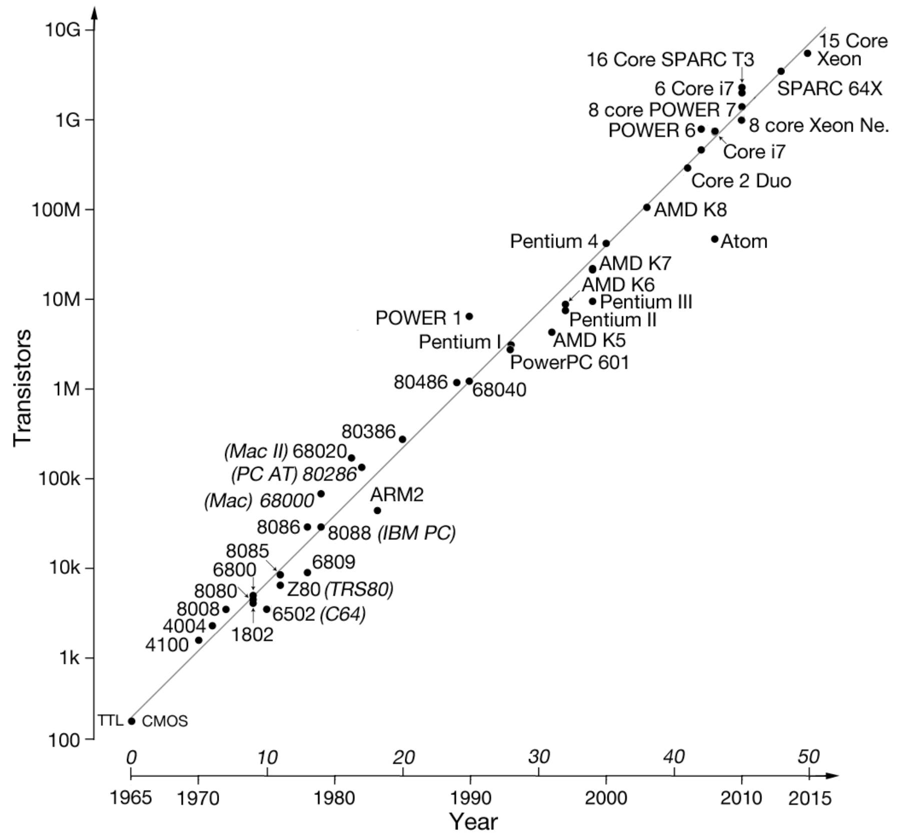


Figure 1.1. Transistor counts in 1965 – 2015. A plot of transistor counts per IC against year of introduction; the vertical line corresponds to exponential growth of transistor count doubling every two years (Scherer 2015).

The performance of SoCs is greatly dependent on the characteristics of the interconnection between the PEs, particularly as the number of PEs in the SoC increases. It is necessary to implement a communication infrastructure that provides the required communication bandwidth between PEs, while keeping latency low and minimising power consumption. Networks on chip have opened a new option for high performance on-chip communication (Dally and Towles 2001). NoCs consist of a chip-wide network of locally connected routers able to relay data from any PE on the network to any other. Compared to the conventional interconnect using shared buses and crossbars, they provide a flexible high performance chip-level communication with regularity and modularity. In the design flow of NoC architectures, the mapping of PEs onto NoC structures plays a crucial role because this affects the traffic flow within the NoC, and can have a significant effect on the overall performance of the system.

This thesis focuses on the implementation of a NoC-based communication infrastructure for a SoC containing specialised hardware for production systems that will implement a cognitive architecture for AGI. Critically, it also proposes an efficient algorithm to map the PEs onto the NoC structure.

1.2 Research motivations

To achieve human-like intelligence, a system must possess a large bank of knowledge. Since the realisation of hard-coded knowledge is slow, tiresome and error-prone, intelligent systems often involve logical expressions of the pattern matching processes, known as production systems, which provide a straightforward mechanism to imply new knowledge using stored information (Nilsson 2014). Whenever the state of the environment changes they perform parallel pattern matching between the current state and remembered information, and trigger appropriate actions. Production systems are, by nature, computation-intensive because of the requirement for continuous parallel repeated searching and matching processes. The integration of a large knowledge base increases the demand on these processes (Hayes-Roth and Jacobstein 1994). However, the immature programming paradigm and tools for them make it harder for a programmer to achieve something in a production system compared with systems programmed using conventional languages. As a result, their performance has, historically, not been satisfactory for complex applications.

Most of the data manipulated by production systems is symbolic because of the ability of symbolic data to represent abstract information, which is one of the important aspects of intelligence (Newell and Simon 1976). Several customised hardware architectures were proposed to improve the efficiency of execution of production systems in the 1980s (Stolfo 1983, Gupta 1985), but these efforts faded away with the improved performance and availability of general purpose computers, and for many years production systems were largely realised on general purpose computers. With the inception of sophisticated intelligent agents, however the execution of production systems on general purpose machines has failed to deliver the performance and speed required in real-time applications. This limitation is expected to become more critical as cognitive architectures evolve towards being able to implement agents that achieve true AGI. However, the massive increase in the number of transistors now implementable on one chip, and the evolution of the MPSoC model, have revived the interest in specialised hardware for large production systems that are subject to real-time time constraints.

Production systems usually spend most of their computation efforts matching the production rules with the state of the environment (Forgy 1979). For a conventional machine, this operation exhausts the available memory and processing resources, and makes the system slow. But the inherently parallel characteristics of production systems are a good match for the large scale parallel computation that can be achieved by multiple PEs integrated on a single IC. The improvement of semiconductor technology provides the ability to realise a hardware-based production system which involves fine-grained parallelism by associating each production rule to a PE, and thus achieving very high higher execution speeds.

NoCs have recently emerged as an effective communication platform for systems with a large number of PEs. They provide on-chip networks of routers to transmit data between PEs. The placement of PEs in the NoC network greatly influences the performance of the system because the communication time, link bandwidth and energy are dependent on it. But determining the optimal mapping of PEs to NoC routers is an NP-hard problem in which the search space increases factorially with the number of PEs and routers. It requires prohibitive computation time to solve using exhaustive algorithms (Garey and Johnson 1979). Therefore, an effective mapping algorithm is required to obtain an near-optimum mapping within a tolerable time limit. Branch and Bound (Lawler and Wood 1966) and Simulated Annealing (Kirkpatrick *et al.* 1983) are two optimisation approaches commonly used to solve this kind of problem. The work described in this thesis examines these approaches to propose an efficient NoC mapping algorithm for hardware-based production systems.

1.3 Thesis objectives

This thesis studies the design of a hardware for production systems, which implements fine-grained parallelism in execution by evaluating production rules in simple homogeneous PEs. The objectives of the research reported in this thesis is to develop an efficient communication platform for the hardware. It targets NoC-based on-chip communication where each router can be associated with one or more PEs. The mapping of PEs to NoC routers is another problem that is also explored in this thesis. Branch and

Bound, and Simulated Annealing based mapping techniques are evaluated in order to propose an improved mapping algorithm. The next Section presents a summary of the original contributions made in the course of addressing these research issues.

1.4 Statement of original contributions

This thesis makes contributions in the area of on-chip communication of homogeneous PEs of cognitive workload. The contributions can be divided into two categories. The first part focuses on the design of a interconnect platform for homogeneous PEs of a hardware-based production system. And the second part concerns the mapping of PEs to NoC routers using optimised mapping algorithms.

1.4.1 On-chip communication of cognitive workload

A conventional computer, with its sequential execution path and centralised memory, is not a good match for the computational requirements of cognitive agents. Rather, these would be better served by hardware that supports fine-grained parallelism and distributed memory, somewhat like the brain. The *Street Processor* is a reconfigurable, flat, parallel computer architecture designed as a SoC with many homogeneous PEs for symbolic cognitive workloads (Frost *et al.* 2015). It has own instruction set, called the *Street language*, to define the behaviour of an agent using production rules.

In this processor, thousands of identical PEs transmit data to each other simultaneously. This subsequently causes new data to be generated by the PEs when their corresponding production rules match the state of the environment, and this results in even more traffic. To make this traffic exchange feasible and efficient, a NoC-based communication platform is developed for this hardware. NoCs have been widely used in embedded heterogeneous SoCs, however no application is reported for event-driven SoCs containing such a large number of homogeneous PEs. Thus the application of NoCs in the class of parallel processors exemplified by the Street Processor is considered to be the principal contribution in this thesis. This work has been presented in *International Workshop on Artificial Intelligence and Cognition (AIC 2015)*, under the title of "A Network-based Communication Platform for a Cognitive Computer" (Numan *et al.* 2015).

1.4.2 Optimised mapping techniques

Optimal mapping of PEs to NoC routers is an NP-hard problem which is a big challenge in NoC design. This work investigates mapping techniques based on two optimisation algorithms. Branch and Bound is a systematic approach, which reduces the search space by deleting unpromising partial solutions without further exploring them. On the other hand, Simulated Annealing is a probabilistic technique, which is designed to avoid the trap of becoming stuck at local optima. A comparative study of the mapping techniques based on these approaches is done in this work to contribute towards determination of a preferred approach of the proposed mapping algorithm.

The NoC mapping process is used as an example of a cognitive problem in this work. In order to support building a self-configurable agent, the Branch and Bound based mapping is itself implemented using Street language. This demonstrates that the Street Processor can dynamically and autonomously re-arrange the placement of PEs based on recent traffic history. The significance of this action is that it shows the capacity of the Street Processor to monitor and improve its own performance.

NoCs are more scalable than other contemporary on-chip communication platforms (Dally and Towles 2004). In this work, the scalability of NoCs is further improved by allowing multiple PEs to be associated with a single router. This consequently reduces the number of routers in a NoC structure and reduces the implementation cost. When the communication patterns include clusters of PEs with a lot of traffic between them, this strategy reduces the inter-router traffic of the network. This approach also shrinks the search space of the PE-to-router mapping problem since a group of PEs (GPE) are assigned to a router, instead of an individual PE. The mapping of GPEs to routers is achieved by an improved Simulated Annealing algorithm, which involves a heuristic initial mapping to expedite the process. This work is in preparation to be submitted to a peer-reviewed journal under the title of "Priority-based Network-on-Chip Mapping of the Processing Elements of Production Systems".

1.5 Overview of the thesis

As outlined in Figure 1.2, this thesis comprises five parts including an overview of the work undertaken, a review on the Street Processor, two parts discussing the above-mentioned contributions and the conclusion. The highlights of each part of the thesis is presented as follows.

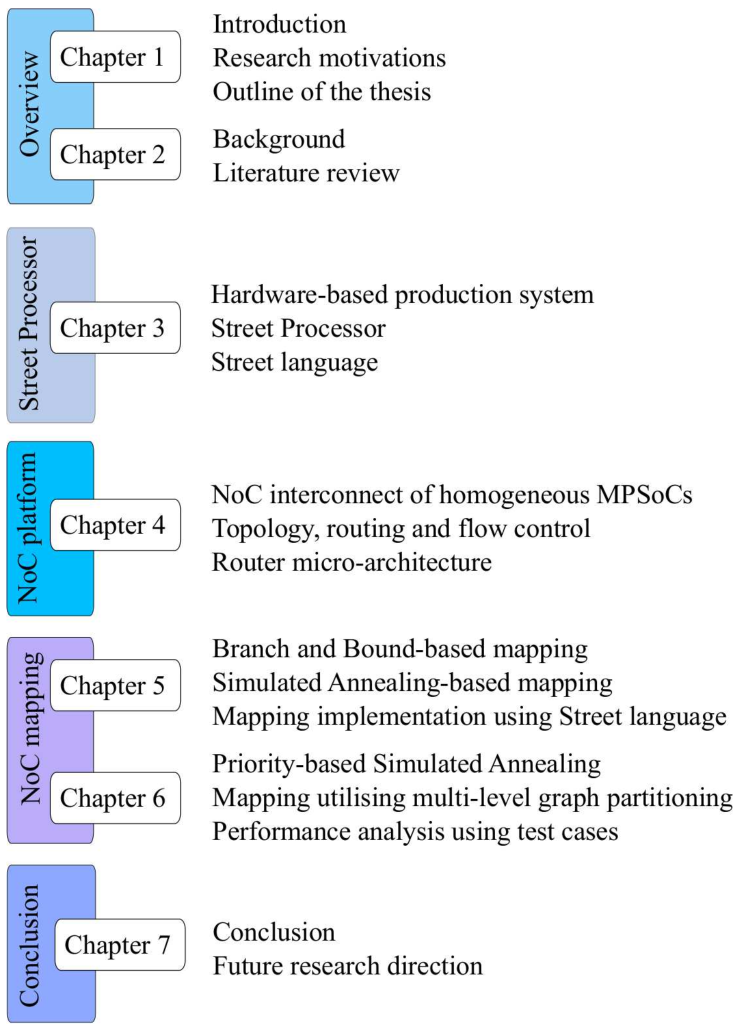

Overview
Chapter 1
Introduction
Research motivations
Outline of the thesis
Chapter 2
Background
Literature review
Street Processor
Chapter 3
Hardware-based production system
Street Processor
Street language
NoC platform
Chapter 4
NoC interconnect of homogeneous MPSoCs
Topology, routing and flow control
Router micro-architecture
NoC mapping
Chapter 5
Branch and Bound-based mapping
Simulated Annealing-based mapping
Mapping implementation using Street language
Chapter 6
Priority-based Simulated Annealing
Mapping utilising multi-level graph partitioning
Performance analysis using test cases
Conclusion
Chapter 7
Conclusion
Future research direction

Figure 1.2. Thesis outline and contributions. This thesis is composed of 7 chapters in total, divided in five major parts. The major contributions lie in the part of NoC mapping.

Overview (Chapters 1 & 2) provides the introductory information, background and a review on the technologies relevant to this thesis. The current chapter has described the motivation, objective and contributions of the work performed in this study. Chapter 2 presents a literature review of the previous works in the fields of cognitive architectures, production systems and NoCs. This chapter also provides justification for the methodologies and approaches used in this work.

Street Processor (Chapter 3) reviews the design of a hardware-based production system. It describes the architecture of the Street Processor, containing many homogeneous PEs. The general structure of PEs are described in this chapter. The production rules are defined by the Street language which is also explained. This chapter also introduces some promising features like sleep period and Big Productors that make the Street Processor a very novel on-chip production system.

NoC platform (Chapter 4) of the SoC with many PEs is presented in this part of the study. First, it presents the justification for using NoCs instead of traditional on-chip interconnect techniques in such systems. Second, a NoC-based communication platform is described for the class of SoCs containing large numbers of homogeneous PEs, represented by the Street Processor. This chapter discusses different aspects of NoC design e.g. topologies, routing schemes, flow control techniques, as well as the micro-architecture of a NoC router.

NoC mapping (Chapters 5 & 6) presents several techniques for mapping homogeneous PEs onto NoC structures. Chapter 5 describes two mapping techniques using Branch and Bound, and Simulated Annealing algorithms. It also demonstrates the comparison between these two techniques. In addition, the Branch and Bound mapping algorithm is implemented using Street language demonstrating the feasibility of constructing self-configurable cognitive agents. In Chapter 6, an improved Priority-based Simulated Annealing algorithm is proposed, which finds an optimum mapping solution in a reasonable computation time without sacrificing mapping quality. The performance of this technique is analysed using two test cases as well as a set of synthetic traffic scenarios.

Conclusion (Chapter 7) summarises the significance of the work, results and contributions presented in this thesis. This chapter also presents an outlook of possible future research work and improvements.

Chapter 2

Background and Related Work

THE use of production systems is very common in cognitive architectures because of their ability to produce effective reasoning behaviour, and simple and regular structure. In spite of being the subject of research for many years, production systems have been limited by their slow performance. Researchers have proposed different algorithms to mitigate this problem and have implemented these algorithms on a variety of hardware systems from conventional computers to systems on chip. SoCs with multiple processing elements have emerged as a promising technology to implement large and complex systems. However, efficient communication between the PEs still poses challenges to researchers. Networks on chip are an efficient solution for on-chip communication, but mapping of PEs onto NoC routers strongly influences the performance of SoCs and is an important area of research. This chapter provides a study on background and related works on cognitive architectures, production systems, NoCs and other related topics to establish the context for the work described in this thesis.

2.1 Introduction

Human-like intelligence in artificial systems, commonly termed as artificial general intelligence in the literature, is a very long studied area of research. In contrast to advanced AI systems such as Deep Blue (Campbell *et al.* 2002) and Watson (Ferrucci *et al.* 2010), which are very expert in specific applications, AGIs are conceived to solve general problems (Thorisson and Helgasson 2012). AGI agents are expected to have human-level intelligence, general knowledge across different domains, the ability to reflect on themselves and the ability to create fundamental innovations and insights (Franklin 2007).

Several approaches have been adopted to achieve AGI in a system, but no one artificial system has been able to approach the level of human intelligence. One of the most promising fields of research directed to this end is cognitive architectures. Cognitive architectures are large systems of heterogeneous modules and components, which operate coherently to solve general problems in multiple domains (Thorisson and Helgasson 2012). As a foundation for AGI systems, they have those characteristics of agents that are common to multiple application domains. They include memories that store information and representation of the agent's goals and knowledge. They also contain well defined processes to interact with the environment, and learning mechanisms to improve their performance over time.

Many cognitive architectures (Forgy 1979, Anderson 1996, Kieras and Meyer 1997) involve production systems as the core of their reasoning because of their ability to describe how humans think. They receive inputs from the environment and compare these against a knowledge base to deliver appropriate responses. They have been generally considered computationally expensive and slow, and hence were largely ignored in industry (Kuo and Moldovan 1992), in spite of their immense promise to support general purpose intelligent systems. But the increasing demand of intelligent systems is continuously pressing to build efficient production systems.

Research on production systems has been pursued in different directions. Most of these efforts considered implementation of production systems on conventional computers. These implementations are prone to under-utilisation of available resources due to the von Neumann bottleneck (Backus 1978). With the continuous evolution of semiconductor process technology, SoCs with multiple PEs have evolved as an efficient solution to this bottleneck for many applications (Jerraya and Wolf 2004). Communication between the on-chip components of MPSoCs has been a broad area of research ever since the MPSoC concept emerged. It is evident that shared bus and ad hoc point-to-point connection based solutions could not support the ever-increasing communication demand of MPSoCs. NoCs, inspired by wide area networks of computers, emerged as a potential solution and opened a new paradigm of on-chip communication (Dally and Towles 2001). The performance of SoCs is influenced by the placement of PEs on the NoC infrastructure. This encouraged researchers to propose efficient algorithms to solve this placement problem.

This chapter starts with a short history of AGI systems in Section 2.2. This is followed by a discussion of cognitive architectures in Section 2.3 that highlights their characteristics and provides a category-wise review of cognitive architectures. This section also briefly describes the two most successful cognitive architectures: Soar (Laird *et al.* 1987, Laird 2012) and ACT-R (Anderson 1996, Anderson *et al.* 2004) , which are both based on production systems. Section 2.4 describes the basic structure of production systems and reports a study on state of the art techniques. Section 2.5 explores SoCs with multiple PEs as the most appropriate implementation technique for complex embedded systems. Finally, Section 2.6 presents the communication model and a review on the architectures of NoCs. The algorithms for the placement of PEs on NoCs structure are also reviewed in this section.

2.2 Artificial general intelligence

An AGI is a system that could successfully performs a wide range of tasks autonomously across different domains in a similar way to a human being. The features that distinguish AGI agents from other AI agents are their abilities to pursue a wide variety of goals embedded in different environments rather than solving a predefined problem. The original aim of AI research was to build human level intelligence. Herbert Simon, who was one of the founders of the concept of AI, forecast about Turing's 'Thinking machine' (Turing 1950) that "... their ability to do these things is going to increase rapidly until – in a visible future – the range of problems they can handle will be coextensive with the range to which the human mind has been applied" (Simon and Newell 1958). AI research started towards this goal, but in 1970s - 1980s, it was realised that the practical problems in achieving this ambitious goal in reality were immense, perhaps intractable. As a consequence, AI researchers cut back their objectives to develop intelligent systems with domain specific targeted solutions, called 'weak AI' to contrast with the 'strong AI' of general artificial intelligence.

Recently, there has been a renewed interest in building human-like intelligent systems. Large corporations like Yahoo, Google, IBM, Facebook are investing in AGI related technologies. Google has acquired 'DeepMind', which builds general purpose learning algorithms for simulations, e-commerce, and games (Shu 2014). Motivated by the human nervous system, IBM introduced self-managed computer systems, called 'autonomic systems', which are able to configure themselves to optimise performance and to protect themselves from attacks (IBM 2005). Yahoo and Facebook are also taking similar interest in this advanced AI technologies (Zacks 2013, Cutler 2014). Government organisations have also begun to develop an interest in strong AI. Intelligent distribution agent (IDA) is an AGI software agent of U.S. Navy that interacts in natural languages and assigns tasks to the pool of personnel autonomously (Franklin 2003).

No particular focus has yet emerged from current AGI research; rather contemporary projects are diverse and often pioneering in nature. This is because, interestingly, there is little consensus on the objective of reproducing 'intelligence' as a whole in computers. Although every approach gets its motivation from the same source, human intelligence, here 'intelligence' is understood in several senses. Consequently, researchers attempt to replicate different aspects of human intelligence. Table 2.1 lists different approaches to achieving human-level intelligence. Among them, cognitive architectures are considered to be one of the most promising approaches to create AGI systems (Pei 2007).

2.3 Cognitive architectures

John Anderson first coined the term 'cognitive architecture' in his book, "The architecture of cognition" (1983). He defined it to be 'the basic principles of operations of a cognitive system'. However, even before he introduced the concept, the idea was implicit in the rule-based information processing theories of Newell and Simon (1972). Newell further elaborated this concept in his book, "Unified theories of cognition" (1990), where he listed a number of important features of cognitive systems including adaptive, dynamic and flexible behaviours, learning, knowledge integration, vast

Table 2.1. Approaches to building AGI agents.

Aspects	Inspiration	Rationale	Examples
Structure	Neuroscience, biology	Since intelligence is produced in human brains, an intelligent computer should simulate brain structure as faithfully as possible	TrueNorth (Merolla *et al.* 2014), SyNAPSE (Srinivasa and Cruz-Albrecht 2012), HTM (Hawkins and George 2006)
Interaction	Linguistics	Intelligence is displayed by effective communication between two parties. For this reason, an intelligent system should interact exactly like a human	Turing test (Turing 1950)
Capability	Domain-specific computer applications	Problem-solving capability proves intelligence. Hence, an intelligent system should be able to solve certain practical problems that are currently solvable by humans only	Deep blue (Campbell *et al.* 2002), expert systems (Jackson 1986)
Behaviour	Psychology	Intelligence is associated to cognitive behaviours that human beings demonstrate, such as perceiving, reasoning, learning and acting. Therefore, an intelligent computer should reproduce these behaviours	Soar (Laird *et al.* 1987), ACT-R (Anderson 1996)
Principle	Logic, mathematics	Intelligence is a form of rationality or optimality. So an intelligent system should always do the right thing according to certain general principles	AIXI (Hutter. 2004), NARS (Wang 1995)

knowledge base and real-time performance. These features were subsequently analysed and partially applied by several architectures (Anderson and Lebiere 2003).

2.3.1 Characteristics of cognitive architectures

All cognitive architectures are expected to demonstrate the ability to analyse the external and internal environment in order to solve both repeating and new problems by learning from experience. The essential characteristics of cognitive architectures are discussed here, some of which are adopted from (Langley *et al.* 2009b).

Identification of events: A cognitive architecture has the ability to identify events and match them with the known or familiar knowledge. This is the initial step in categorising events to determine appropriate responses to them. A cognitive architecture supports this feature by representing patterns of events. Production systems (Forgy 1979, Miranker 1987) of cognitive architectures perform this task through the use of conditions in their production rules.

Selection from alternatives: Often, multiple patterns in the knowledge base can be matched by the events. A cognitive architecture selects the most appropriate one from these alternatives. This decision making, known as *conflict resolution*, involves several criteria depending on architectures (Luger 2004, Miranker *et al.* 1990). A good cognitive architecture refines these criteria based on its experiences.

Problem solving strategy Human-like intelligent agents are most likely to handle new situations. This requires plans and problem solving mechanisms (Langley *et al.* 2009a). A cognitive architecture represents a plan as an ordered set of actions, their expected effects, and the manner in which these effects enable later actions.

Remembering and learning: A cognitive architecture encodes and stores the results of cognitive processing in memory, usually referred to as *episodic memory*, to retrieve them when required. An ideal cognitive architecture should also have some ways of learning to improve its performance by generalising specific experiences (Laird 2012).

Knowledge sharing: A cognitive architecture should be able to communicate with others agents so that they can obtain and share knowledge (Ferguson and Allen

Table 2.2. Classifications of cognitive architectures.

	Memory type	Representative architecture
Symbolic	Rule-based memory	Soar (1987), EPIC (1997)
	Graph-based memory	SNePS (2007)
Connectionist	Globalist memory	IBCA (2000)
	Localist memory	Cortronics (2007)
Hybrid	Localist-distributed memory	CLARION (2006)
	Symbolic-connectionist memory	ACT-R (1996)

2011). This may also require a way for transforming the knowledge from internal representations to a form suitable for communication.

Occasional uncertainty: Like human beings, a cognitive architecture may allow occasional uncertainties in its operations. This allows it to explore new experiences which might result in it learning new things. This type of learning is considered by some researchers to be an essential feature required to achieve cognition (Sun 2001).

2.3.2 Related work on cognitive architectures

In this section, cognitive architectures are reviewed and compared from different perspectives (Thorisson and Helgasson 2012, Langley *et al.* 2009b, Duch *et al.* 2008). Knowledge representation is considered a key property that can be used to differentiate cognitive architectures. An architecture achieves knowledge from the information stored in its memory. Various types of memory keep information about the environment, current activities and previous experiences. The importance of memory has been demonstrated from different perspectives in the recent literature (Hecht-Nielsen 2007, Hoya 2005, Hawkins and Blakeslee 2004). Based on the types of memories, cognitive architectures can be broadly divided into three categories: *symbolic*, *connectionist* and *hybrid* architectures, as summarised in Table 2.2.

Symbolic cognitive architectures

Symbolic architectures focus on information processing in a top-down analytic approach using high level symbols or declarative knowledge. They have the ability to input, output, store and alter symbolic entities, and to carry out appropriate actions to reach their goals. Soar (Laird *et al.* 1987), EPIC (Kieras and Meyer 1997), ICARUS (Langley *et al.* 2009a) and SNePS (Shapiro *et al.* 2007) are some prominent symbolic cognitive architectures. In symbolic architectures knowledge can be embedded in production systems as production rules (Laird *et al.* 1987). Production rules provide a flexible and context dependent representation of knowledge with their conditions matching the current state and actions modifying it. Production systems are discussed in more detail in Section 2.4. The knowledge can also be represented using directed graphs like semantic networks, conceptual graphs and schemata (Sowa 1992, Minsky 1974).

Connectionist cognitive architectures

Connectionist architectures have been widely used to solve domain specific problems. Less commonly they have also been used as the basis for general cognitive architectures. These architectures view knowledge as being encoded into simple neuron-like processing nodes. The nodes use low level activation signals flowing over a network that interact with each other in a specific way changing their internal states and revealing interesting emergent properties. The multi-layer perceptron and other neural networks based on delocalised transfer functions process information in a distributed, global way (O'Reilly and Munakata 2000). All parameters of such networks influence their outputs. On the other hand, the basis set expansion networks that use localised functions are examples of localist networks (Blachnik and Duch 2008, Duch *et al.* 2001). The output signals for a given input depend only on a small subset of units that are activated. This kind of architecture has been criticised in the literature for not demonstrating the desired functionalities of cognitive architectures (Langley *et al.* 2009b). IBCA (O'Reilly and Munakata 2000) and Cortronics (Hecht-Nielsen 2007) are two examples of connectionist architectures.

Hybrid cognitive architectures

Hybrid architectures result from combining the symbolic and connectionist paradigms of cognitive architectures. Symbolic architectures are able to process information and realise high level cognitive functions, whereas connectionist architectures are better suited for capturing context specific behaviour and handling many pieces of low level information simultaneously. The potential benefit of a combined approach is therefore to have each method address the limitations of the other, allowing creation of a complete intelligent system architecture that covers all levels of cognitive processing. The localist-distributed class of hybrid architectures comprises a combination of localist modules with each concept specified by one processing node and distributed modules with each concept represented by a set of overlapping nodes. In comparison, the symbolic-connectionist class involves a mixture of symbolic modules (rule or graph based memory) and connectionist modules (localist or distributed type) (Sun and Alexandre 2013). ACT-R (Anderson *et al.* 2004), CLARION (Sun 2006) and DUAL (Kokinov 1994) represent the most successful hybrid cognitive architectures.

Many of the proposed architectures in the literature have been abandoned due to their limitations, while others have been vigorously pursued. Soar and ACT-R are the most successful and mature cognitive architectures. The following sections briefly discuss these two architectures.

2.3.3 Soar

Soar (State, Operator And Result), one of the most successful symbolic cognitive architectures, has been gradually developed and extended over the last three decades (Laird *et al.* 1987, Laird 2012). It is a typical example of rule-based architectures designed to model general intelligence. Soar stores its procedural knowledge in the form of *production rules*, arranged in terms of operators. The *operators* act in the problem space that is the set of *states* in order to achieve the goal or *result*. Figure 2.1 shows the block diagram of the Soar architecture (Laird 2012).

The current version of the Soar architecture includes different types of memories that interact during its processing cycle. These are the short term *working memory*, and long term *semantic*, *procedural* and *episodic memories*. These memories are briefly described here:

Working memory stores information about the current state of the environment. The working memory elements (WMEs) are either supplied by sensors or copied from other memories based on relevancy to the present state. Working memory also contains an activation mechanism indicating the relevance and usefulness of WMEs.

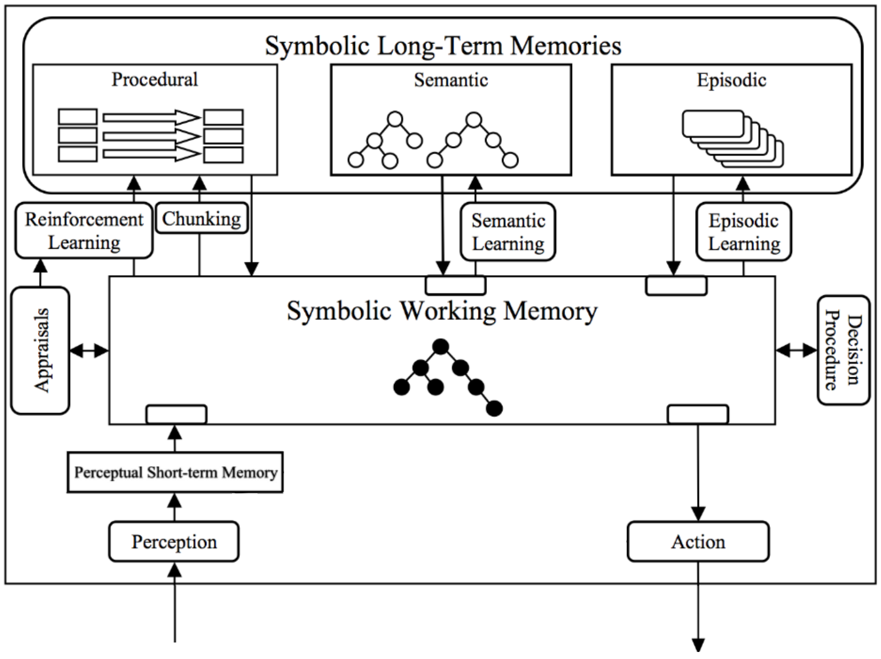


Figure 2.1. Block diagram of the Soar architecture. (Laird 2012)

Semantic memory keeps declarative information and facts about the problem space. Knowledge is retrieved from semantic memory via deliberate associative retrieval i.e., using appropriate cues. Most of the knowledge is initialised from an existing knowledge base and the rest is acquired when new information is learned.

Episodic memory stores the memory episodes that a Soar agent has experienced. It includes contextualised information about specific events. This provides the agent with the ability to use prior experiences to handle the current state.

Procedural memory stores the production rules that support the proposal, evaluation, selection and application of operators during processing cycles. The operators act on the problem space and make changes to working memory.

The execution of a Soar agent is governed by its *processing cycle* in which the agent chooses an appropriate operator based on the information of its working memory and production rules. This cycle can be broken into five phases as shown in Figure 2.2. In the input phase, Soar receives the changes of current state to the perception block and a perceptual buffer in working memory is updated. In the next phase, the current state is elaborated by firing the production rules in parallel that match the changes to retrieve information relevant to that state. Then the operators are proposed by production rules for the current state. Once operators are proposed, other rules register preferences for them based on different criteria such as frequency or recency of rule firings. The operators are then evaluated based on preferences by a fixed decision procedure, and if a single operator is unambiguously preferred by the evaluation criteria, it is selected for application. In the application phase, the selected operator is applied to make changes to the state in order to move the agent closer to its goal. Finally, in the output phase, Soar sends information of state changes to problem space via its actuators.

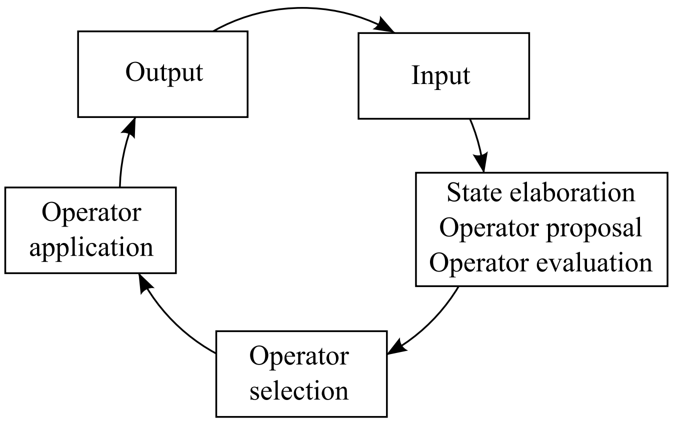


Figure 2.2. Soar processing cycle.

The Soar processing cycle repeatedly proposes, selects, and applies operators to a state. However, when knowledge about operator selection is insufficient to uniquely determine the appropriate operator to apply, an *impasse* occurs. When an impasse occurs, Soar decomposes the state into a *substate* that includes the reasons for the impasse and the state that the impasse arose in. From the perspective of the new state, the earlier state is known as the superstate. In the substate, operators are proposed, selected and applied in the same way as the superstate. Additional impasses may be encountered in a substate. This may lead to a stack of substates. Soar applies operators on states in the substates to resolve the impasse. When it is resolved, all the subsequent substates below it disappear and Soar learns new information. The reason for the impasse and its actions are then converted to production rules which summarise the processing that resolved the impasse. This learning process is known as *chunking* (Laird *et al.* 1986). The new production rules generated by chunking are added to the procedural memory to be used directly in similar situations in the future without creating substates.

A variety of complicated agents have been developed using Soar. The most visible has been TAC-Air-Soar that is used for training fighter pilots of U.S. Army (Tambe *et al.*

1995). Soar is also used in interactive computer games (Magerko *et al.* 2004), modeling human language processing (Lewis 1993) and categorisation (Miller and Laird 1996).

2.3.4 ACT-R

ACT-R (Adaptive Control of Thought-Rational) is a hybrid cognitive architecture that is concerned primarily with modelling human cognition (Anderson 1996, Anderson *et al.* 2004). It was developed based on a human associative memory (HAM) model (Anderson and Bower 1973) which was later extended into the model of human cognition (Anderson 1983). ACT-R architecture is composed of *modules*, *memories*, *buffers* and a *pattern matcher*. Figure 2.3 shows the high level architecture of ACT-R.

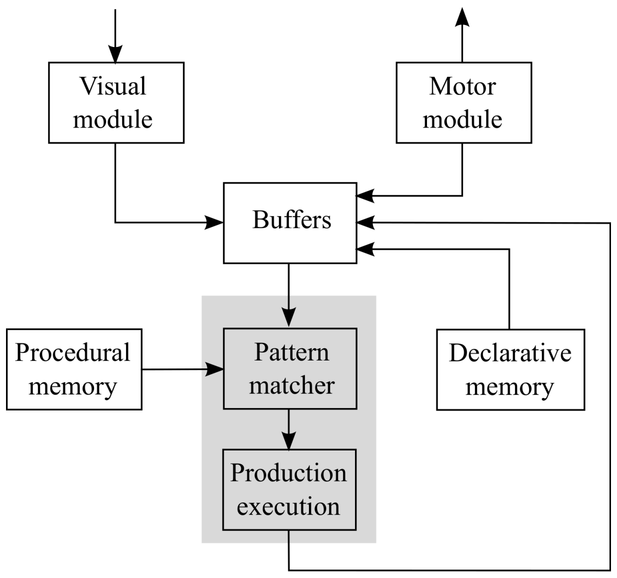


Figure 2.3. Block diagram of the ACT-R architecture. (Budiu 2013)

Modules: ACT-R is organised into a set of modules, each of which processes a different type of information to take care of the interface with the environment. The most well developed modules are sensory modules for visual processing and motor modules for actions.

Memories: There are two kinds of long term memories in ACT-R: *declarative memory* encodes factual knowledge about the environment into chunks (different from 'chunks' in Soar); and *procedural memory* coordinates the processing of the modules using its production rules. Both are realised as a symbolic-connectionist structure, where the symbolic level consists of production rules and chunks, and the sub-symbolic level of a massively parallel connectionist structure.

Buffers: ACT-R accesses its modules through buffers. For each module, a dedicated buffer serves as the interface with that module. The contents of the buffers collectively comprise the short term memory.

Pattern matcher: The pattern matcher searches for a production rule that matches against the contents of short term memory. Only one such rule can be executed at a given moment. That rule, when executed, modifies the buffers and thus changes the state of the system.

Each declarative chunk has an associated base activation that reflects its past usage and influences its retrieval from long term memory, whereas each production rule has an expected cost and probability of success. These enable an analytic characterisation of connectionist computations to measure the general usefulness of a chunk and production rule. These parameters influence the rule matching process of the pattern matcher that searches for a production rule to match against the state of the buffers. If several rules match, ACT-R computes the utility for each matched rule as the difference between its expected benefit and its expected cost. The system selects the production rule with the highest utility and executes its action. That action alters the buffers and thus changes the state of the system. This change causes new production rules to be matched and results in even more changes. Some changes modify existing structures, whereas others initiate actions in the associated modules, such as executing a motor command or retrieving a chunk from long term declarative memory.

ACT-R learns declarative knowledge through problem solving. The base activation for declarative chunks increases when they are used by production rules, and decreases otherwise. Similarly, the cost and success probability for productions is updated based on their observed behaviour. The architecture can learn entirely new rules from sample solutions through a process of production compilation that analyses dependencies of multiple rule firings, replaces constants with variables, and combines them into new conditions and actions (Taatgen 2005).

Over the years, a number of applications have been developed based on ACT-R model for understanding a variety of phenomena from the experimental psychology literature, including aspects of memory, attention, reasoning, problem solving and language processing. Anderson related ACT-R modules to different areas of the brain and developed models that match results from brain-imaging studies (Anderson 2007). It has been used in natural languages (Budiu and Anderson 2004), complex problem solving

(Byrne and Kirlik 2005), tutoring (Koedinger *et al.* 1997) and controlling mobile robots that interact with humans (Trafton *et al.* 2005).

2.4 Production systems

Soar, ACT-R and many other cognitive architectures such as EPIC (Kieras and Meyer 1997), CLARION (Sun 2006) and NARS (Wang 1995, Wang 2006) use production systems as a core mechanism to achieve their goals. A production system is one whose knowledge base contains the domain knowledge encoded in the form of rules to mimic human reasoning. It stems from the observation that much of human reasoning can be expressed by a set of *if-then* or *condition-action* rules. Instead of representing knowledge in a declarative and static way, a production system represents it in a general way to handle different situations.

2.4.1 Basic structure of production systems

A production system generally consists of three components: a set of *production rules* to define behaviours, a global knowledge base, referred as *working memory*, representing the environment and a control mechanism called the *inference engine*.

Production rules consist of a left hand side (LHS) of conditions and a right hand side (RHS) of actions. If the condition part is matched with the current state of the environment, then the action part is executed. Each rule represents some knowledge about the problem solving process.

Working memory is the total knowledge base of the current state of the environment. It is represented as a set of attributes defining the domain world. The attributes are called WMEs, which are symbolic triples (e.g. (ID1, HasA, Tail) arranged in a graph data structure.

Inference engine The inference engine is a sequential program that controls the processing of the production system. It has three steps: rule matching (*match*), conflict resolution (*select*) and rule execution (*act*). The inference engine is said to be operating in forward chaining when the system starts in an initial state and executes production rules to reach the final state. A backward chaining system starts at the goal state and tries to find the possible conditions that resulted in that state. Backward chaining is often used for learning, while forward chaining is common in reasoning (Rich and Knight 1991). In this research, the inference engine is considered in forward chaining only.

The schematic of Figure 2.4 shows the execution cycle of a production system that works on production rules and working memory, and is dictated by the steps of the inference engine.

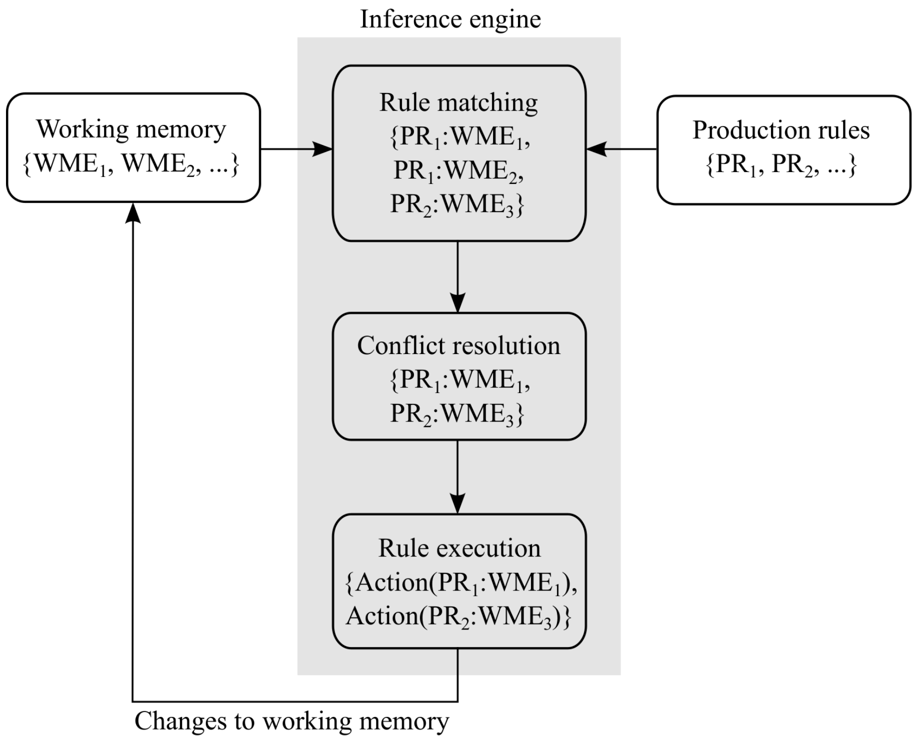

Inference engine
Working memory {WME1, WME2, ...}
Rule matching {PR1:WME1, PR1:WME2, PR2:WME3}
Production rules {PR1, PR2, ...}
Conflict resolution {PR1:WME1, PR2:WME3}
Rule execution {Action(PR1:WME1), Action(PR2:WME3)}
Changes to working memory

Figure 2.4. Processing cycle of a production system. $PR_1 : WME_1$, $PR_1 : WME_2$ and $PR_2 : WME_3$ are three rule instantiations. $Action(PR_1 : WME_1)$ and $Action(PR_2 : WME_3)$ are the actions for the instantiations $PR_1 : WME_1$ and $PR_2 : WME_3$ respectively.

Rule matching: When the WMEs in working memory match the conditions of a production rule, the rule is *instantiated*. The rule may match multiple sets of WMEs at the same time, resulting in a *conflict set* of instantiations. For example, if PR_1 is satisfied by both WME_1 and WME_2, then there are two instantiations of PR_1 and they create a conflict set.

Conflict resolution: In conflict resolution, a single rule instantiation is selected from the conflict set based on a variety of criteria. The criteria may be simple, such as selecting the fist rule activated in the match phase, or may involve complex rule selection heuristics. In many cases, the activation is selected according to the recency of the matched WMEs in the working memory. This comes from the assumption that more recent matches are more likely to be relevant to the agent's current situation (Luger 2004, Miranker *et al.* 1990). It is also intended to guide the agent towards following a single line of reasoning. In Figure 2.4, PR_1 is selected for execution in the conflict resolution step.

Rule execution: Once a rule instantiation is selected, it *fires*, i.e. the action part of the production rule executes. This makes changes to working memory by adding, removing or modifying WMEs. The changes in working memory affect new rules and cause the cycle to begin again.

2.4.2 Related work on production systems

There is a long history of research on production systems. They were extensively investigated during the 1980s and 1990s (Kuo and Moldovan 1992) but they were considered to be computationally expensive and slow due to the underlying matching process. This encouraged the researchers to improve the performance and resource managements of production systems in different ways. The research efforts on production systems can be classified into three categories: *sequential matching*, *parallel matching* and *parallel rule firing* algorithms.

Sequential matching algorithms

Rete (Forgy 1979) is the most popular sequential matching algorithm and has been considered as a standard technique since its development. Forgy initially developed Rete for the OPS5 production language (Forgy 1981). Rete compiles the conditions of production rules in the form of an augmented dataflow network (Miranker 1987), and the new WMEs added to working memory are sequentially matched with these conditions. The partial matchings of these conditions are referred as *tokens*. The tokens are stored in the memory nodes of the Rete network. This matching algorithm performed slowly during modifications of WMEs, because the modifications were performed by deletions followed by additions of WMEs to working memory. When the previous WME was deleted the states of the network was updated. During addition of the WME with a different value, many tokens that were removed previously, were re-instantiated and this caused redundant network updates. TREAT (Miranker 1987) addressed this problem by using a more efficient network update procedure. The deletions in TREAT were faster than those of Rete, but the additions resulted in time-consuming operations due to exhaustive searches to determine complete matches of the conditions. However, it was claimed that the speed-up during deletions was much greater than the loss during additions (Miranker 1987). Again, the problem of redundant network updates of the Rete algorithm was addressed by sharing the network nodes, but the effectiveness of this procedure was criticised by later works (Schor *et al.* 1986). The problem of ineffective network sharing was addressed by YES/RETE (Schor *et al.* 1986) by introducing a direct WME modification operation, instead of the conventional add-after-delete operation. This was reported to be almost 10 times faster than the Rete algorithm.

Parallel matching algorithms

To reduce the execution time of matching operations, Gupta (1985) proposed a parallel architecture of the Rete matching algorithm using the states saved in intermediate memory nodes. He suggested partitioning the Rete network into several processes and assigning the processes to multiprocessors. A shared memory architecture was suggested for this solution to allow the data structures be shared amongst multiple concurrent processes. He expected 100-fold execution speed-up on the original Rete algorithm, however the results showed only a 10-fold speed-up, because only a very small number of production rules were affected by the changes in working memory and there was a large variation in the processing requirements of the affected rules. The potential speed-up achievable by this implementation was limited by the number of available processors. DADO (Stolfo 1983) overcame this limitation by using a large set of processing nodes, each associated to an individual production rule. The processing nodes were interconnected to form a complete binary tree. The performance of the DADO prototype was evaluated by measuring the execution times of a set of benchmark programs (Stolfo 1987) which reported 2 to 31-fold speed-ups compared to the system running a compiled version of OPS5. However, DADO had a drawback of low utilisation of hardware parallelism (Kuo and Moldovan 1992). Moreover, both DADO and Gupta's parallel matching solution suffered from cross-product effects, which referred to the case when an incoming token found several tokens to be matched, and as a result of which a large number of tokens were sent to the successor of that node.

Parallel rule firing algorithms

A second class of parallel production systems involved parallel rule firing techniques. These techniques attempted to increase the available parallelism by parallelising not only the match phase, but all phases of the processing cycle of production systems. As these production systems did not have any sequential conflict resolution criteria, the instantiations of the rules that were not dependent on each other, were only allowed to fire concurrently in a production cycle. Two types of parallel rule firing systems are described in the literature: one that applies a parallel conflict resolution strategy and the other that allows non-deterministic executions. In the first type of systems the parallel conflict resolution strategies are encoded as a set of meta-rules (Stolfo *et al.* 1991). The inputs to these meta-rules are the instantiations that are already matched. By careful analysis of these meta-rules, a set of non-interfering rule instantiations is selected and fired. In non-deterministic production systems (Ishida 1990, Gamble 1990, Pasik 1989), a rule instantiation is chosen randomly and fired without the help of any conflict resolution strategy. Since this random selection leads to many possible solutions of a particular problem, some of these systems leave the responsibility of correctness of a particular solution to the programmers, whereas the others consider the context of the problem to apply instantiations of the production rules.

Production systems were the subject of a large body of work in the 1980s and early 1990s. The research on a specialised hardware for production systems were abandoned due to lack of computing capacity and communication bandwidth of hardware technologies. The current advancements in chip technology motivates many of the ideas to be revisited. The next section will review SoCs with multiple PEs as a suitable technology to realise production systems.

2.5 Multi-processor systems on chip

For several decades the performance of computation systems has been increasing exponentially, following the prediction of Gordon Moore who anticipated that the number of transistors in a single chip would increase two fold every two years (Moore 1965, Moore 1998). However, to achieve the full benefits of this transistor growth per chip, manufacturers had to face the challenge of a long persisting problem – the von Neumann bottleneck (Backus 1978). The transfer mechanism between memory and processor limits the performance of the architecture, especially in modern systems with very high clock rates. Therefore, no matter how fast the processor is, or how big the memory is, the performance of the system is constrained by the rate at which instructions and data are moved back and forth from the memory to the processor. This effect can be mitigated to a certain extent by employing caches and pipelines, but it cannot be eliminated (Hennessy and Patterson 2011). As this problem is embedded in the architecture itself, some experts believe that this should be solved by moving to a completely new architecture rather than adding new features to it (Mang *et al.* 2009).

One of the possible directions to this end is to build a system that integrates all components of a digital system into a single chip. This architecture is called a SoC. They are currently viewed as the most suitable implementation technique to build complex embedded systems that provide high performance in terms of both timing and power. SoCs with multiple programmable PEs or cores, known as multi-processor SoCs have emerged as an important class in the past decade (Jerraya and Wolf 2004). They combine several embedded processing cores, memories and specialised circuitry (accelerators, peripherals) interconnected through a dedicated infrastructure to provide a complete integrated system. MPSoCs are widely used in the application of networking, communications, signal processing and multimedia.

MPSoCs have two distinct branches in the literature: *heterogeneous* and *homogeneous* MPSoCs, although the 'MPSoC' acronym generally refers to the heterogeneous class.

Heterogeneous MPSoCs contain PEs with differing functionalities, such as general purpose processors, digital signal processors (DSPs), memories, accelerators and peripherals, reflecting the need of the expected application domain. To take full advantage of heterogeneous MPSoCs, the system software must use the execution characteristics of each application to predict its processing needs and then schedule it to PEs that match those needs. The predictions can minimise the performance loss to the system as a whole rather than that of a single application. C-5 (Freescale 2001), Nexperia (Dutta *et al.* 2001) and OMAP (Texas Instruments Inc. 2004) are some of the leading heterogeneous MPSoCs architectures.

Homogeneous MPSoCs introduce parallelism through replication of many identical PEs placed in a regular fabric. The PEs run multiple instructions in parallel, increasing overall speed for programs amenable to parallel computing. This approach has a major positive consequence of making system design a matter of instantiation capability instead of architecture complexity. The homogeneous style is generally used for data-parallel systems. The Lucent Daytona architecture (Ackland *et al.* 2000), a pioneer of this class, was designed for wireless base stations.

A homogeneous approach is more scalable but less power efficient, while a heterogeneous approach offers the best power vs performance trade-off (Saponara and Fanucci 2012). Due to their good power efficiency, heterogeneous MPSoCs are used for portable systems and more generally embedded systems. These systems typically run a well-defined set of software applications and hence the processing resources can be optimised to suit the software workload. Homogeneous approaches are commonly used for computers, servers and supercomputing, where the hardware must achieve high performance over a wide variety of different software workloads (Azulsystems 2016, Picochip 2012).

2.6 Networks on chip

Following Moore's law, the transistor density is increasing every year (ITRS 2015), however uniprocessor architectures are not increasing in performance proportionately. On the contrary, MPSoCs are are emerging as the prevailing architecture for both general purpose and application specific applications. As the number of PEs increases and the system becomes more complex, the need for a scalable on-chip communication infrastructure that can deliver high bandwidth is gaining more importance.

The communication structures of earlier SoCs were characterised by custom designed ad hoc mixes of buses and point-to-point links (Lahiri *et al.* 2001). Buses were well understood concepts and were easy to model, but in a highly interconnected MPSoC, they became a communication bottleneck. Moreover, the power usage per communication event increased due to higher capacitive load caused by the increased number of PEs. A crossbar can overcome some of the limitations of the buses but is not scalable and so was considered as an intermediate solution only. Dedicated point-to-point links were optimal in terms of bandwidth availability, latency, and power usage as they were designed especially for this given purpose. Although they were simple and easy to design and model, the number of links increased exponentially with the number of PEs, and impose an increasingly large area overhead.

For systems with a small number of PEs, ad hoc communication structures might be viable, but as the systems grew and the design cycle time requirements decreased, the need for more generalised solutions became pressing. The bus and crossbar based network structures consumed far more energy than desirable to achieve the required on-chip communications and bandwidth (Dally and Towles 2001). For maximum flexibility and scalability, it was required to move towards a shared, segmented global communication structure. The search for alternative architectures led to the concept of NoCs (Guerrier and Greiner 2000, Dally and Towles 2001, Benini and Micheli 2002). The key concept behind NoCs was to use a network of routers to enable data flow between the PEs associated to routers more efficiently, and to allow simultaneous communications over multiple channels. This solved the issues of energy and performance inefficiencies incurred by shared bus communications. Such a distributed communication fabric scaled well with chip size and complexity. This also helped to improve aggregate performance by exploiting parallel operations at different segments of the network structure.

2.6.1 NoC communication model

The basic idea of NoCs was borrowed from the wide area networks of computers. A seven layer open systems interconnection (OSI) model (Zimmermann 1980) was proposed to interconnect heterogeneous and distributed systems. This layered structure decomposes the communication problem into more manageable components at different hierarchical layers. Each layer solves one part of the problem. In addition, layering provides a more modular design. At each layer, protocols and services, which are implementation independent, are well defined. The nodes residing in the same layer can thus communicate with each other transparently. Adding new services to one layer only needs to modify the functionality at that layer only, reusing the functions provided at all the other layers. Due to these advantages, several NoCs researchers (Millberg *et al.* 2004, Sgroi *et al.* 2001) have followed this model and adapted it to build a protocol stack for on-chip communication.

Inspired by the OSI model, NoC architectures can be partitioned into four layers: *system layer*, *network interface layer*, *network layer* and *link layer*. Figure 2.5 shows the flow of data through the layers of NoC structure.

System layer defines PEs and their operations. At this layer, most of the network implementation details are hidden from the PEs. They transmit data as if they are directly connected to each other.

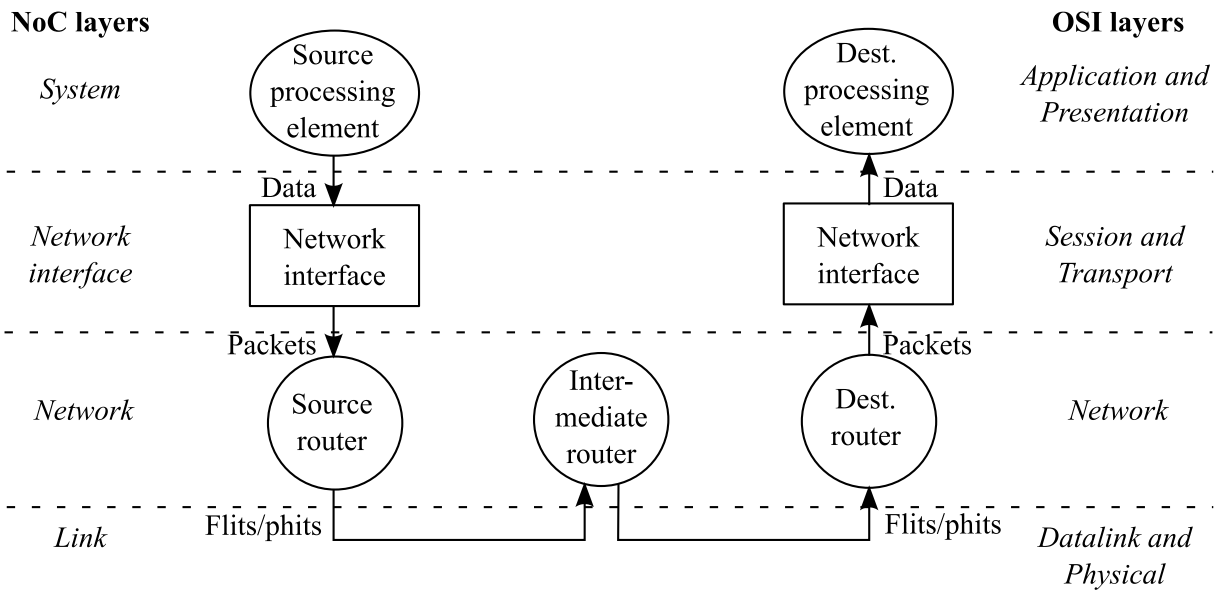


Figure 2.5. Data flow through different layers in NoC and OSI models.

Network interface layer decouples the PEs from the network. It handles the end-to-end flow control, encapsulating the data generated by the PEs for the routing strategy of the network. The data is broken into *packets* which may contain information about their destination.

Network layer consists of the routers or switches and links of the NoC structures. An on-chip network is defined mainly by its topology and its protocol. *Topology* concerns the layout and connectivity of the routers and links on the chip. *Protocol* dictates how these routers and links are used. This layer is described in more detail in Chapter 4.

Link layer consists of one or more channels which can be either virtual or physical. It works with *flits* (flow control unit) that are subdivisions of packets. The flits may be again divided into *phits* (physical units) which are the minimum size datagram that can be transmitted in one link transaction. In most cases flits and phits are equivalent.

Compared to wide area networks, in NoCs, the layers are generally more closely related to each other. NoCs can be designed based on knowledge of the PEs to be connected and the known characteristics of the traffic to be handled. This awareness provides additional advantages to NoCs designers when building an optimised on-chip communication infrastructure.

2.6.2 Related work on NoC architectures

Efficient on-chip communication mechanisms have been an active research topic for more than a decade (Moraes *et al.* 2004, Bjerregaard and Mahadevan 2006). Sonics Micronetwork (Flynn 1997) was an example of an evolutionary solution which generalised the on-chip interconnect to support higher bandwidth than shared buses. ST-BUS (Bona *et al.* 2004) was another example of this class, which provided the designers flexibility to instantiate both shared bus or crossbar interconnect configurations. These architectures provided higher bandwidth than simple buses, but the wiring delay and scalability problems of these architectures encouraged researchers to propose more versatile solutions. In order to achieve this goal, researchers moved to a NoC paradigm for on-chip communication.

A number designs addressing different aspects of NoCs have been proposed over the years. Æthereal (Rijpkema *et al.* 2003b) was a design that aimed at providing a complete NoC infrastructure for heterogeneous SoCs with end-to-end quality of service guarantees. The network supported guaranteed throughput for real-time applications and best effort traffic for timing-unconstrained applications. Support for heterogeneous architectures required highly configurable NoC building blocks, customizable at instantiation time for a specific application domain. Xpipes interconnect (Dall'Osso *et al.* 2012) and its synthesiser XpipesCompiler (Jalabert *et al.* 2004) instantiated an application specific NoC from a library of composable soft macros e.g. network interfaces, links and switches, that were highly parametrizable and provided a reliable performance.

NoC platforms with regular topologies are suitable for homogeneous SoCs. NOSTRUM (Kumar *et al.* 2002), proposed by researchers at KTH in Stockholm, adopted a grid-based, router driven communication platform for on-chip communication. The scalable programmable integrated network (SPIN) (Guerrier and Greiner 2000) was another regular network architecture. It adopted cut-through routing to minimise latency and storage requirements. The SoCBUS (Wiklund and Liu 2003) used a two dimensional mesh network with packets routing through the network while locking the circuit as they traversed.

After several years of intensive research, there are now several credible NoCs in the market. Sonics SiliconBackplane (Wingard 2001) was the first commercial NoC. It offered a time division multiple access style interconnection network. Arteris (Fanet 2005) and Silistix (Martin *et al.* 2010) were two other examples of commercial NoCs.

The Silistix interconnect was an asynchronous interconnect that had adaptors for common bus protocols and was meant to fit into a SoC built by using the globally asynchronous locally synchronous (GALS) paradigm.

2.6.3 Related work on NoCs mapping

The mapping of PEs to NoC routers is an important step of NoC design. It directly impacts the performance of SoCs, because the communication time, required link bandwidth, admissible delay and power consumption of routers are influenced by the placements of PEs. A number of mapping techniques have been reported recently in several surveys (Sahu and Chattopadhyay 2013, Pop and Kumar 2004). In (Sahu and Chattopadhyay 2013), the authors categorised NoC mapping algorithms into two broad classes: *static* and *dynamic* algorithms. In static mapping the association between PEs and routers is determined before execution and this is not changed thereafter, whereas in dynamic mapping, this association may change during execution based on network traffic conditions. Static algorithms are further divided into several sub-classes. Among them *deterministic* and *heuristic* algorithms are the principal ones.

Dynamic algorithms

Dynamic mapping algorithms update the assignments of PEs to routers during execution of the system. They monitor the traffic to detect performance bottlenecks and distribute the traffic over the network. The works in (de Souza Carvalho *et al.* 2010, Chou and Marculescu 2008, Chou *et al.* 2008) represent some leading mapping techniques of this class. As these algorithms depend on the current traffic load, they are expected to result in better solutions. However, the computational overhead of the mapping algorithms increases delay and energy consumption of the systems at run time.

Static deterministic algorithms

Deterministic algorithms are search-based static mapping techniques. The Branch and Bound mapping algorithm (Hu and Marculescu 2005) belongs to this class. It is a systematic search algorithm that topologically finds the mapping by searching the solution in tree branches and bounding unallowable solutions. It results in an energy and performance aware mapping for tile-based regular NoC architectures to satisfy the specified design constraints through bandwidth reservation. Based on the Branch and Bound algorithm, a traffic balanced mapping algorithm (TBMAP) for two dimensional mesh was proposed in (Lin *et al.* 2008). To ensure balanced traffic over the network, it sacrificed minimum paths between PEs in some cases. In (Reshadi *et al.* 2010), the Branch and Bound algorithm was applied with heuristic-based NMAP mapping (Murali and Micheli 2004). It performed better than both of the algorithms in terms of communication cost, power consumption and network latency. Branch and Bound algorithms have drawbacks in the form of large memory and long CPU time requirements, and hence, are suitable for smaller problems (Sahu and Chattopadhyay 2013).

Static heuristic algorithms

Heuristic algorithms are suitable for NP-hard problems that require a prohibitively long time for finding exact solutions. The objective of this approach is to produce a solution in a reasonable time frame that is good enough, if not the best among all possible solutions. Mapping of PEs to the NoC routers is an example of such problems. Heuristic mappings based on genetic algorithms (Lei and Kumar 2003, Jena and Sharma 2007), particle swarm optimisation (Fekr *et al.* 2010, Wenbiao *et al.* 2007) and ant colony optimisation (Wang *et al.* 2011) are some evolutionary techniques that are inspired by natural phenomena. These kind of mapping techniques usually suffer from slow rate of convergence. In contrast to evolutionary techniques, constructive heuristic algorithms are much faster. They generate partial solutions sequentially, and at the end deliver the final mapping solution. These algorithms can be constructive without iterative improvement, such as PMAP (Koziris *et al.* 2000) and SMAP (Saeidi *et al.* 2007), or constructive with iterative improvements, such as NMAP (Murali and Micheli 2004), SUNMAP (Murali and De Micheli 2004) and Simulated Annealing based mapping (Harmanani and Farah 2008). Simulated Annealing is an optimisation algorithm that probabilistically accepts bad results to avoid the trap of converging to local minima while searching for a near optimal solution. It is discussed in more detail in Chapter 5.

2.7 Chapter summary

This thesis describes a NoC for a homogeneous MPSoC designed for execution of a cognitive architecture. This chapter has discussed the background and research related to the topics covered in this thesis. It started with a discussion of AGI as the motivation of this work. Cognitive architectures are considered to be amongst the most prominent approaches to achieving AGI in a system. They effectively replicate cognitive behaviours through the activities of perceiving, reasoning, learning and acting. Researchers have taken several strategies to realise these cognitive activities on systems. Production systems are very commonly used as the core of reasoning in intelligent systems but have the limitation of being slow due to the mechanism used to match the state of the environment with its knowledge base. A number of algorithms have been proposed to speed up the matching procedure. Most of these algorithms considered realisation of production systems on conventional computers that cannot efficiently utilise the current advancement of semiconductor technologies. SoCs with multiple PEs emerged as a very promising system implementation technique to take advantage of these advancements. On the other hand, the on-chip communication greatly impacts the performance of SoCs. Researchers in this field have adopted NoCs as the most viable platform for on-chip communication. In order to support the next chapters of this thesis, this chapter reviewed the research activities on NoCs, especially the assignment of PEs to NoC routers.

Chapter 3

Hardware-based Production System

THIS chapter presents an overview of the *Street Processor*, a new processing architecture for symbolic cognitive workloads. This processor serves as the platform for the author's research on communication infrastructure for hardware-based production systems. The processor has its own instruction set, called *Street language*, to define the behaviour of an agent using production rules. The Street Processor supports fine-grained parallelism by implementing the production rules in simple, identical processing elements. However, it has some unique synchronisation techniques to control its parallelism when required. The Street Processor also implements some innovative concepts such as sleep period and Big Productors, which have the potential to improve its performance for cognitive applications.

3.1 Introduction

Production systems provide intelligence to a system by defining a set of rules based on the current state that act when there is a change to the current state. They have been the subject of a large body of work during the 1980s and early 1990s (Kuo and Moldovan 1992). They have often been used in research on intelligent systems including expert systems (McDermott 1980, Waterman 1986) and cognitive models (Anderson 1983, Laird *et al.* 1987, Newell 1990). However, despite their long history, production systems are computationally expensive.

The quest for an efficient production system has led to efforts in different research directions. One of the most important contributions was the development of efficient production rule matching algorithms. These are used to determine when a rule is satisfied by the current state. The invention of the Rete algorithm (Forgy 1979) was a major step in this direction. Since then, a number of improvements and modifications to Rete have been proposed (Miranker 1987, Schor *et al.* 1986). Another important advancement towards this end, was the OPS5 system (Forgy 1981) which was the first rule-based language to be used successfully in expert systems. However, it was criticised because its naive rule matching technique occupied most of its computation efforts (Gupta 1985). Gupta's research focussed on improvements to rule matching algorithms and transforming existing algorithms to exploit parallel conventional hardware. The DADO computer (Stolfo 1983) was a notable example of the processors that emerged from this work.

The imperative for improved algorithms and optimised hardware to support production systems waned with the explosion of performance and affordability of conventional computing hardware during the last three decades, which have enabled other approaches to implementing AI. However, the cognitive agents are becoming more complex, and this trend is expected to accelerate as we strive towards agents that achieve true artificial general intelligence. With the growth in complexity of cognitive agents and greater accessibility of efficient application-specific hardware, specialised hardware architectures for cognitive computing holds significant promise of improved performance over conventional computing hardware, especially in embedded cognitive applications. This chapter describes a new kind of hardware, named the *Street Processor*[1], as an efficient platform for production systems. The goal of the Street project

[1]The *Street* project is an outcome of a team of researchers, of which the author is a member. This work is presented in (Frost *et al.* 2015).

is to design a computer architecture that takes advantage of the huge number of transistors available in modern integrated circuits to achieve advanced cognitive computation in real-time, but with much lower power dissipation than current computers. The Street Processor is quite different from conventional computers. It realises fine-grained parallel execution using hardware that distributes computation and memory among many PEs. It shares these features with other recently announced cognitive computers (Merolla *et al.* 2014, Kumar 2013); however it is optimised for processing at a higher level of abstraction, using production rules and symbols rather than neuron models and synaptic weights.

This chapter starts with the description of two popular matching algorithms, in Section 3.2. The design of the Street Processor is tightly linked with the design of its instruction set, which we call *Street language*. For the sake of clarity, the Street language is presented first in Section 3.4, with little reference to the underlying hardware. The Street Processor architecture is described in Section 3.5. Then, different aspects of the Street Processor are discussed in Sections 3.6 and 3.7.

3.2 Production systems

A production system is a model of computation that has been widely used in AI. This can be achieved by providing the system a symbolic representation of its environment and of behaviours the agent is expected to exhibit in response to changes in its environment. This representation is then systematically manipulated until a solution is produced (Newell and Simon 1976, Laird *et al.* 1987). The organisation and execution characteristics of a production system is illustrated in Section 2.4.

The major computational cost of a production system is matching the rules against working memory. A naive approach that compares all the conditions with all the working memory elements in each cycle spends 90% of its execution time searching for matches (Forgy 1979). As the number of rules is increased the processing cost quickly exhausts available computational resources. Several matching algorithms have been developed to avoid this problem (Oflazer 1986, Raschid *et al.* 1988, Miranker *et al.* 1990), but Rete (Forgy 1979, Forgy 1981) and TREAT (Miranker 1987, Miranker and Lofaso 1991) are considered the most prominent among them.

3.2.1 Rete matching algorithm

Forgy (1979) observed that the actions of a fired rule affected only a small proportion of the working memory in each cycle. So rather than re-evaluating the entire production system in each cycle, he proposed the Rete matching algorithm, which only computes incremental changes to the set of satisfied rules due to working memory changes. This increases the processing speed and makes the production system more efficient. Rete maintains some intermediate memories of all partial matches. The algorithm processes the changes in working memory to update the intermediate memories and determine which production rules completely or partially match the new state of working memory. Together with the fact that in the usual cases there are only a few memory changes in each cycle, and those changes affect only a small number of production rules, the Rete matching algorithm works efficiently with a large number of production rules.

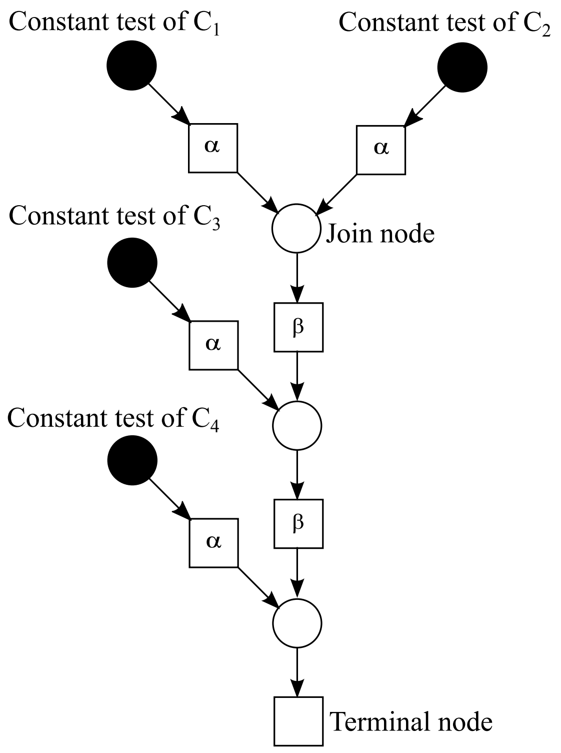
The Rete algorithm compiles the conditions of production rules into a discrimination network. The partial matches of WMEs with the conditions are stored in intermediate *alpha* and *beta* memory nodes of the network. Constant tests that check the existence of constant symbols in WMEs are performed, and the WMEs, which passes the constant tests are stored in alpha memories. The alpha memory nodes are connected to two-input *join nodes* that find the partial binding between conditions. Beta memories store partial instantiations of the rules i.e. combinations of WMEs that match some but not all of the conditions of a production. These partial instantiations are called *tokens*.

As an analogy to relational databases, the working memory can be considered as a 'relation' and the production rule as a 'query'. The constant tests represent a 'select' operation over WMEs. For each condition, there is an alpha memory that stores the result of that 'select' operation. Let us consider a generic production rule having conditions C_1, C_2, C_3 and C_4, in Figure 3.1 (a). The complete matching of production rule PR with working memory is given by $match(C_1) \bowtie match(C_2) \bowtie match(C_3) \bowtie match(C_4)$, where the symbol $\bowtie$ is a natural join operator between two conditions. The join nodes do these joins, and each beta memory stores the results of one of the intermediate joins.

Whenever there is any change in working memory, this is checked using constant tests and the appropriate alpha memories are updated. These updates are propagated over to the attached join nodes, activating those nodes. If any new partial instantiations are created, they are added to the appropriate beta memories and then propagated down of the network, activating other nodes. Whenever the propagation reaches the bottom of the network, it flags that all the conditions are completely matched. This is

```
PR {
      (C1)      //conditions
      (C2)
      (C3)
      (C4)
-->
      (A1)      //actions
      (A2)
}
```

(a)



(b)

Figure 3.1. An example of Rete discrimination network.

done by a terminal node that signals the newly found complete match and sends the rule bindings to the conflict set. Figure 3.1 (b) shows the Rete discrimination network corresponding to the production rule shown in Figure 3.1 (a).

3.2.2 TREAT

There is a trade-off in the amount of information that should be saved during each Rete matching cycle. Saving all information requires many intermediate memories and processing becomes time consuming if working memory changes very frequently. On the other hand, saving too little information may result in unexpected matchings. Miranker (1987) pointed out that beta memories cause overhead in memory management. Every time a WME is removed, matching must be performed to find those entries that need to be deleted. To solve this problem, he proposed TREAT (Miranker 1987) eliminating beta nodes from the network. Unlike alpha memories in Rete, the memories are broken into three partitions: *old*, *new-delete* and *new-add*. The old partition contains the partially matched WMEs of the previous cycle. In the current cycle, the new WMEs are added to the memories of the new-delete and the new-add partitions depending on the current operation.

When a new WME is added, an exhaustive search through the network is performed to determine new rule bindings. As a result, additions in TREAT are slower than those of the Rete algorithm. When a WME is deleted, it is deleted from the alpha memory, and its instantiations in the conflict set are also deleted. Therefore, deletions in TREAT are faster than those of the Rete algorithm. Miranker has shown that in many cases the speed-up obtained in deletion is greater than the loss in addition (Miranker 1987). It is claimed that TREAT outperforms Rete, often by more than 50% (Miranker and Lofaso 1991), though Nayak's outcome contradicts the claim as he thought that some issues like the long chain effect and handling of combinational logics are overlooked in TREAT (Nayak *et al.* 1988). This debate opens an avenue of further investigation on the TREAT matching algorithm.

3.3 Street Processor: Hardware-based production system

The word 'Street' is inspired by the names of the 'Soar' cognitive architecture and the 'Rete' matching algorithm. However, the Street Processor architecture significantly differs from both of them. It is a reconfigurable, flat, parallel architecture designed for processing symbolic cognitive workloads. It is designed for use in real-time embedded implementations of artificial general intelligence, exemplified by the plethora of potential autonomous robotics applications. It leads to a hardware architecture that is very different from conventional computers, consisting of many simple PEs, called *productors*, executing and communicating in parallel.

The Street Processor executes a parallel production language directly on it. This language, called the Street language, is inspired by OPS5 (Forgy 1981) and the languages used in the Soar (Laird 2012) and ACT-R (Anderson 1996) cognitive architectures. However, the Street language is different from all of these. The language is described in the next section.

A Street Processor is based loosely on the Rete matching algorithm with Miranker's alternative approach, TREAT. The Street language leads to an efficient TREAT-like rule matching hardware. It is asynchronous, with no global match-select-act cycle as found in other production systems, explained in Section 3.2.1. This asynchronous model provides the best opportunity to parallelise traditional production systems at the application level (Amaral and Ghosh 1993). In the work that follows all Street code is executed on a simulator developed in the Java platform (Phillips 2016).

3.4 Street language

The Street language is used to express a set of production rules that implement an intelligent agent. Like other production systems, each rule is an *if-then* statement: *if* a specified pattern (condition) exists in working memory, the rule fires, i.e. *then* the rule executes (action) to make changes to working memory. The production rules operate concurrently and asynchronously. Consequently the language is event-driven and non-deterministic.

The OPS5, Soar and ACT-R production systems use a set of *conflict resolution* methods to select a single rule to fire at some point in an execution cycle. OPS5 uses a fixed set of rules to choose a single instantiation to fire (Forgy 1981). Soar uses a system of *operators* and *preferences* to allow the runtime execution to influence the selection (Laird 2012). ACT-R uses a utility value mechanism, which assigns a metric to rules (Bothell 2004). The Street language has no inherent execution cycle or conflict resolution; any kind of strategy or decision-cycle may be implemented as desired using rules.

It is inefficient for all of the productors to concurrently access the same shared working memory, and just as bad for each to have its own complete private copy of working memory. The Rete matching algorithm addresses this by dividing working memory into the alpha memories. Similarly, in a Street Processor, each productor only stores that subset of working memory that might eventually match its *if* conditions. When a productor makes a change to working memory, it communicates the change to the other affected productors. Hence the system is entirely event driven. A change to working memory can trigger a cascade of further changes. This supports the design of Street language to map production rules to hardware following the commonalities of other production languages.

3.4.1 Working memory

Following other production systems, the working memory in Street represents the current state of the system. It is a set of working memory elements (WMEs). Each WME has one or more fields called *attributes*. Attributes are represented internally as 64 bit words. They can represent signed integers or hashed strings. For instance, the WME `(ID17 source ID2)` has 3 attributes: `ID17`, `source` and `ID2`, where `ID17`, `source` and `ID2` are strings. Here is a simple example of working memory of just 3 WMEs:

```
{(ID17 name Torrens), (ID17 source ID2), (isCounted ID5)}
```

3.4.2 Production rules

Each Street production rule consists of one or more condition elements (CEs) and actions. Each CE and action contain any number of attributes. In the example of Figure 3.2, `(<p> type dog)` and `(<p> age (<a> > 7))` are CEs and `(<p> isOld)` is an action. The rule searches working memory for two WMEs with matching first attributes, one of them having second and third attributes matching the strings `type` and `dog`, and the other WME having a second attribute matching the string `age` and a third attribute with a numeric value greater than 7. If it finds these two WMEs, it fires and the action and creates a new WME with first attribute equal to the first attribute of the CEs, and a second attribute equal to `isOld`. The interpretation of this production rule is that it searches working memory for records of dogs with ages greater than 7, and labels them as old. A complex cognitive agent would consist of hundreds or thousands of production rules operating on symbolic and numeric data in working memory.

```
st {oldDogs                 // a rule begins with 'st{' and a name
    (<p> type dog)          // a rule may have one or more CEs
    (<p> age (<a> > 7))     // CEs are enclosed in brackets
-->
    (<p> isOld)             // there may be one or more actions
}                           // a rule ends with a '}'
```

Figure 3.2. A Street production rule.

Condition elements

A condition element (CE) specifies WMEs to be matched in the working memory. The CE `(<p> type dog)` matches any 3-attribute WME where the second and third attributes are `type` and `dog` respectively. During the matching operation, a *constant test* checks for these constants in WMEs. The first attribute is assigned to a variable <p>, which can be any value. A *variable test* checks this variable for consistent variable bindings in WMEs. A subset of working memory that satisfies all of the CEs in a production rule with consistent variable assignments is called an *instantiation* of the rule. So instantiations of the rule above will be pairs of WMEs in working memory such as:

```
{(pet1 type dog), (pet1 age 8)}
```

Note that the first attributes must be the same as they were specified by the same variable <p>. It is said that these WMEs are *joined* on any shared variables.

As demonstrated in `(<p> age (<a> > 7))`, CEs can test binary relations between attributes and constants. In this example, a match occurs only if the third attribute has value greater than 7, in which case variable <a> is assigned that value.

The special relation `(<x> != <y>)` is also allowed. This specifies that the corresponding attribute must join with any appearances of the variable <x> in other conditions, and must not be equal to the variable <y> bound in any other condition. For example, in Figure 3.3, the CEs check if <p> has <a> and <b>, and both are different. In this case, <p> has at least two items.

```
st {hasAtLeastTwoItems
    (<p> has <a>)
    (<p> has (<b> != <a>))
-->
    (<p> isHappy)
}
```

Figure 3.3. Example of a inequality check.

It is also possible to search for the absence of a WME in memory using a *negative condition element*. For example, in Figure 3.4, `-(<p> isToy)` checks if there is no WME that declares that <p> is a toy. If both the CEs of the rule are true, <p> is labelled `isReal`.

```
st {realDogs
    (<p> type dog)
    -(<p> isToy)
-->
    (<p> isReal)
}
```

Figure 3.4. Example of a negative condition element.

However, the Street language does not allow other relations between variables, such as `(<x> > <y>)`. This is because variable matching is performed in parallel in hardware using content addressable memories (CAMs). Variable relations other than equality and inequality in this scheme incur substantial hardware overhead. It is more efficient to subtract <y> from <x> in one rule, and compare the result against 0 in another.

Actions

The actions of a production rule are performed for each instantiation of the rule. They may only add or delete WMEs. Actions such as `(<p> isOld)` and `(<p> isReal)` in Figures 3.2 and 3.4 add WMEs to working memory. Variables in actions that appear in the rule's CEs take on the values from the rule's instantiation; variables that do not appear on the CEs are given a new, unique identifier value. Actions can also remove WMEs from working memory. For example, in Figure 3.5 `-(<s> counter 4)` deletes the WME from working memory when the condition is true.

A single change to working memory can instantiate multiple production rules in which case the actions for all of these new instantiations will be executed. For example, if the alpha memories of the rules of `oldDogs` in Figure 3.2 and `realDogs` in Figure 3.4 already contain the WMEs `(pet2 age 9)` and `-(pet2 isToy)` respectively, and a new WME `(pet2 type dog)` arrives in working memory, both the rules will fire in parallel and create the two new WMEs `(<p> isOld)` and `(<p> isReal)`.

```
st {resetCounter
     (<s> counter 4)
-->
    (<s> counter 0)
    -(<s> counter 4)
}
```

Figure 3.5. Example of a negative action.

Moreover, a single change to working memory can create multiple instantiations of a production rule. For instance, the working memory currently contains the WMEs `(pet1 type dog)` and `(pet2 type dog)`, and a new WME `(startRunning true)` arrives, then it will cause two instantiations of the rule `runningDog` of Figure 3.6, and `(pet1 isRunning)` and `(pet2 isRunning)` WMEs will be added to the working memory.

```
st {runningDog
     (<p> type dog)
     (startRunning true)
-->
    (<p> isRunning)
}
```

Figure 3.6. Example of multiple instantiation of a production rule.

Actions can include logic and arithmetic operations between variables and constants or variables and variables. For example, `(<n> difference (<x>-<y>))`.To simplify the requirements on underlying hardware, they are currently limited to elementary integer arithmetic, boolean and bitwise logic.

In some other production languages, the actions of a production rule are undone when the rule no longer matches. Soar has two kinds of rule: one that makes persistent changes to working memory, and another that only applies while its instantiation is true (Laird 2012). In Street language all rules make persistent changes to working memory. A WME added to working memory will remain there until it is explicitly deleted. This simple model makes it possible to realise both kinds of Soar rule, although a rule that makes only temporary changes to working memory will need the support of additional rules to delete WMEs it creates.

3.4.3 Synchronisation

Because Street production rules are executed concurrently and asynchronously, it is possible for non-deterministic behaviour to arise due to race conditions. Here it is expected that occasional non-determinism will often be tolerated as a normal aspect of cognitive computing. In cases where determinism is important, Street language provides mechanisms to avoid and resolve races.

Bundled actions: Actions from a given rule and corresponding to a given instantiation are grouped together as bundled actions. They will always be executed in the order in which they are written on the RHS, and will not be interleaved with other actions of the same rule. Hence the rule in Figure 3.7, will always add the WME `(<p> barks)` to memory. If the two conflicting actions, to remove `(<p> barks)` and to add `(<p> barks)`, appeared in two different rules that were instantiated at the same time, then the outcome would be undefined.

```
st {barkingDogs
    (<p> type dog)
    (<p> isReal)
-->
    -(<p> barks)
    (<p> barks)
}
```

Figure 3.7. A rule to demonstrate bundled actions.

Sequencing rules: To resolve races, Street language provides a special sequencing rule, indicated by the use of a modified arrow, `~~>`. If an action (or bundled action) creates multiple new instantiations of a sequencing rule, it will only fire its actions for one of those instantiations. Which instantiation it chooses to fire is arbitrary. It will not fire again for any instantiation until the rule is reset. A sequencing rule is reset when there is no match in working memory for any of its CEs.

To clarify the need for sequencing rules, consider the production rule in Figure 3.8. This rule is intended to count the number of WMEs in memory of the form `(ID1 dog <i>)`. It works fine so long as WMEs of this form arrive in working memory one by one, with sufficient delay between them to instantiate the rule for each WME change.

```
/* it is assumed that (ID1 count 0) is created initially */
st {countDogs
    (ID1 dog <i>)
    (ID1 count <c>)
-->
    -(ID1 dog <i>)    // don't count it again
    -(ID1 count <c>)  // remove the old count
    (ID1 count <c>+1) // increment the count
}
```

Figure 3.8. A Street production rule susceptible to race conditions.

Problems occur when dogs arrive simultaneously or even just quickly one after another. Imagine `(ID1 dog fred)` is created, quickly followed by `(ID1 dog ... tom)`. The rule will fire with `<i> = fred` and `<c> = 0`. There is no guarantee that these changes will take effect before the rule fires again with `<i> = tom` and `<c> = 0`. The result will be that both 'fred' and 'tom' will be removed from working memory, but the count will only be 1, instead of 2. The delay between an action and its effect on working memory is an inevitable consequence of the distributed hardware structure.

The race in Figure 3.8 can be resolved by replacing the `-->` with `~~>` so that it becomes a sequencing rule. This time, the rule will fire for `<i> = fred` and `<c> ... = 0` as before; however, it will not fire again when `(ID1 dog tom)` is created, even though this creates a new instantiation. But then the rule is notified of the action `-(ID1 count 0)` and, assuming there are no other `(ID1 count <c>)` WMEs in memory, the rule will be reset. When the final action from counting 'fred', `(ID1 count 1)` is processed, it will create a new instance of the rule with `<i>` ... `= tom` and `<c> = 1`. After its actions are taken both the 'fred' and 'tom' will have been removed from memory, and the count will become 2 as expected.

3.5 Street Processor architecture

The parallel Street language is directly executed in the Street Processor. Figure 3.9 shows the Street Processor alongside a conventional computer system when both are executing an agent using a symbolic cognitive architecture. The Street Processor has been designed specifically to support cognitive processing. As indicated in the figure, it consists of a large number of hardware processing elements or productors, with a single production rule assigned to each[2]. The productors are connected via interconnect networks and communicate using tokens to notify each other of changes to working memory. Rather than broadcasting tokens to all productors, a static *destination table* is determined for each productor by analysing the constant symbols of the production rules. The destination table lists the productors affected by the action of the current productor. This destination table is also used to initially allocate production rules to productors. This allocation remains static during execution, but is updated when the Street Processor *sleeps* periodically (discussed in Section 3.7).

3.5.1 Micro-architecture of a productor

A Street Processor contains a number of simple identical productors. A simplified block-diagram of a productor is shown in Figure 3.10. Each productor contains *configuration registers*, a *production match controller*, a block of *alpha memory* and an Arithmetic Logic Unit (ALU). Investigating the implementation issues of the productor are still unresolved and matters for further research. They are also not within the scope of this thesis, hence not discussed in broad detail, rather a very brief description is provided.

Configuration registers store the definition of the associated production rule, i.e. the CEs and actions of the rule.

Production match controller is a state machine that coordinates the productor's match-act cycle. For each incoming token it updates the contents of the alpha memories,

[2]It is possible for a production rule to spread over multiple productors. This is discussed in Section 3.5.2

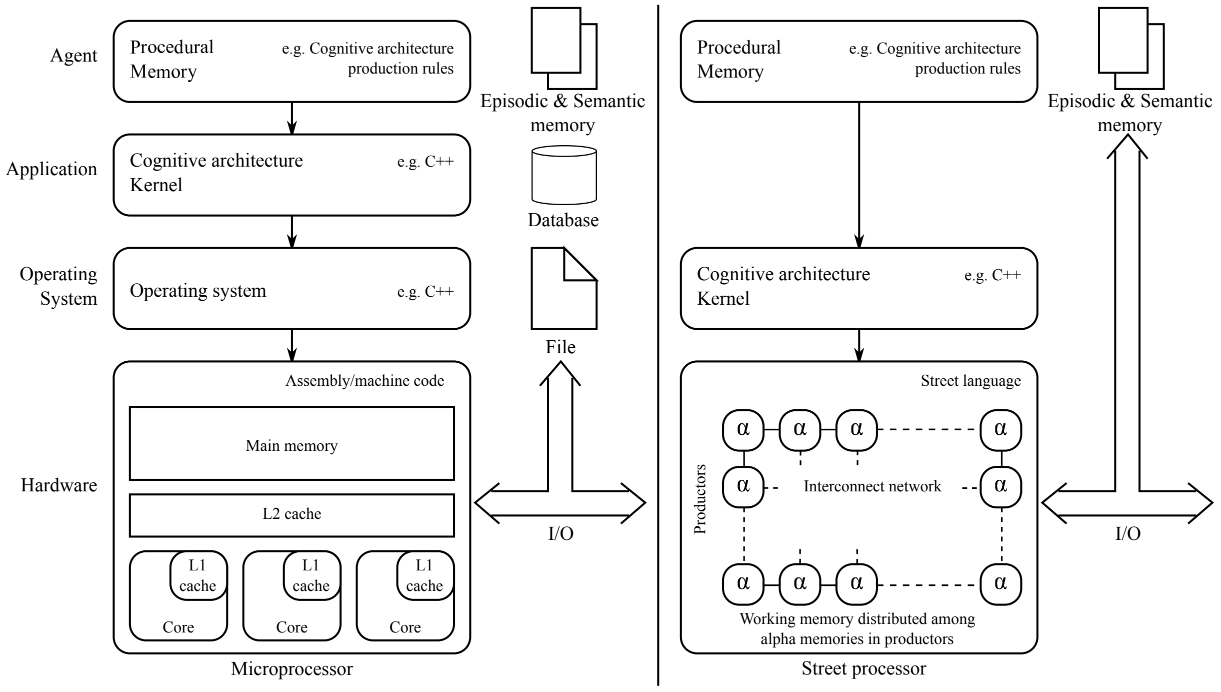


Figure 3.9. Hardware and software stack on a conventional computer and the Street Processor.

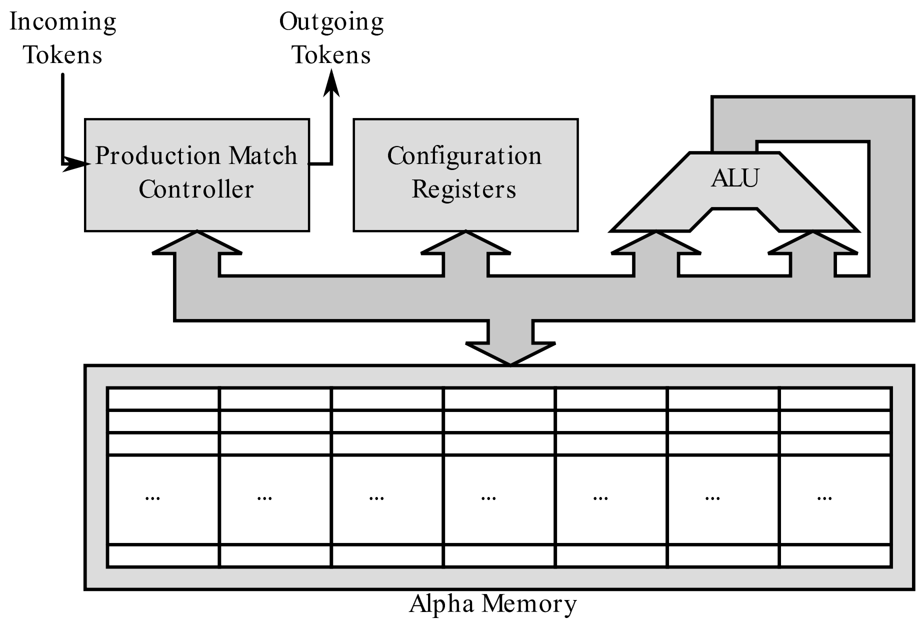


Figure 3.10. Block diagram of a productor.

finds new instantiations (*match*), and outputs tokens corresponding to the rule's actions (*act*).

Alpha memory comprises subsets of working memory matching each CE. It is a block of modified CAM called Relational CAM (RCAM)[3]. The RCAM is used to accelerate searching by matching alpha memory cells in parallel to find new instantiations of a rule. It has a 64 bit column for every variable in the rule's CEs. It is assumed that this column size is large enough to handle a large number of variables in each production rule. The depth of an alpha memory and the number of alpha memory columns per productor are design parameters, and the best choice will depend on the workload. The RCAM includes a *selected* flag in each row. Different search operations can be performed using this flag to select or deselect rows that match the cue, which represents the attribute values in WMEs. The contents of the first selected row can be read from the array and deselected so that iteration continues through the set of selected rows. The hardware overhead for this extra RCAM functionality is trivial compared to the area required for the CAM array, yet it provides exactly the operations required for the matching algorithm.

3.5.2 Big Productor

When a production rule cannot fit in a single productor, either because it has a large number of CEs and variables (and needs too many columns of alpha memory), or it matches a large number of WMEs (the alpha memory requirement exceeds the capacity of a productor) during runtime, the flat architecture of a Street Processor allows the productor to spread over multiple productors, making a *Big Productor*. One of the productors of the Big Productor acts as the master and its configuration registers and production match controller stays active. It is considered as a source or destination of tokens, and the Big Productor uses the destination table attached to it. The other productors of the Big Productor support the master productor by lending their alpha memories. The network routers associated with them are used to forward the tokens only. If a Big Productor is created during runtime, the supporting productors may reside some distance from the master productor. This may affect the system performance due to communication latency, but the situation will be resolved during the next sleep period when the distant productors are re-mapped closer to the master productor. Chapters 5 and 6 investigate different mapping techniques in more detail. When

[3]This work is in preparation to be submitted to a peer-reviewed journal under the title of "Relational Content Addressable Memory".

the mapping technique involves GPEs (which will be discussed in Chapter 6), all of the PEs of the Big Productor are mapped to a single router to reduce communication latency. Figure 3.11 shows an example of a Big Productor where p_4 works as a master productor and is considered as a source or destination of tokens. The router r_4 associated to p_4 stays active and the routers r_1 and r_2 forward tokens only.

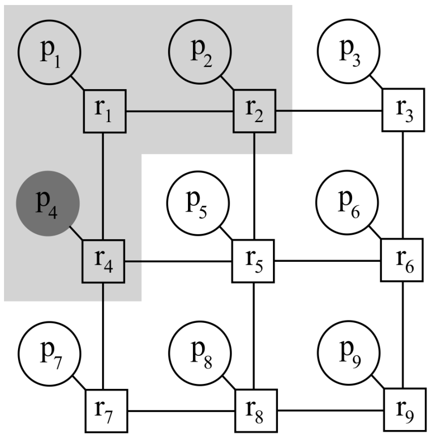


Figure 3.11. An example of a Big Productor.

3.6 Dependency graph

The actions of a production rule make changes in the working memory. If the changes can cause another rule to be instantiated, then the second rule is said to be dependent on the first rule. The degree of dependency is determined by the number of actions of the first rule that causes the second rule to fire. For example, in Figure 3.12, the production rule `oldDogs` adds `(<p> isOld)` to working memory, which may cause `tiredDogs` rule to be instantiated. Hence, `tiredDogs` is dependent on `oldDogs` with the degree of dependency equal to 1.

This dependency provides a good indication of traffic between productors. The Street Processor keeps track of the runtime traffic between productors for a given period of time. It generates a *dependency graph* using the traffic statistics and this is used to refine productor placements later on.

The mapping of productors onto an interconnect network depends on the dependency graph (discussed in Chapters 5 and 6). It is a weighted directed graph, where each vertex represents an active productor. The weighted edges between vertices represent total traffic over a given period of time. This graph is used to map the productors onto

```
st {oldDogs
    (<p> type dog)
    (<p> age (<a> > 7))
-->
    (<p> isOld)
}
st {tiredDogs
    (<p> type dog)
    (<p> isOld)
    (<p> workingHour (<h> > 2))
-->
    (<p> isTired)
}
```

Figure 3.12. An example of dependencies between two production rules.

the interconnect network so that the most inter-dependent productors are placed close together in the expectation that this will reduce communication latency and power consumption.

3.7 Sleep period

Depending on the changes in working memory, different production rules fire at different rates. It is also usual that a production rule fires at different frequencies in different periods of time. The Street Processor stores traffic statistics due to rule firings during runtime. At regular intervals (expected to be of the order of 24 hours) or when the utilisation of alpha memories exceeds a certain threshold, the Street Processor will pause its normal execution of reacting to the changes in working memory. During this time it uses the recorded traffic statistics to update its dependency graph. It evaluates the dependency graph to update the assignment of production rules to productors so that highly inter-dependent rules are assigned to productors close together, to minimise the overall network traffic, latency and power consumption. This is analogous to human sleeping (Landmann *et al.* 2014), and is termed the *sleep period* of the Street Processor.

When the Street Processor wakes up after a sleep period, it continues to operate as before, but with performance that will improve because of the revised rule-to-productor mapping if, as hypothesized, the re-mapping process is effective. Sub-optimal placement may adversely affect the performance of the Street Processor until the next sleep, increasing latency and power consumption, but does not stop it operating correctly.

Re-assignment of production rules to productors involves changing the content of the configuration registers for each productor. Each productor has a single associated network router, which uses a static destination table to direct the tokens to productors that may be affected by them. During the sleep period, the memory contents and destination tables are also modified. However, the network structure remains unchanged. The mechanism to transfer rule definition and memory contents between productors is still under investigation. We envisage that this can be done by using a dedicated supervisory processor, or by the Street Processor itself using additional production rules. The production rules to define the mapping between production rules and productors based on traffic record will be discussed in detail in Chapter 5.

3.8 Conclusion

The Street Processor is a new kind of hardware architecture that has been designed especially for cognitive computing. It executes a parallel production language directly in hardware with the aim of realising advanced cognitive agents more power-efficiently than conventional computers. The language is much simpler than OPS5 and the languages used in Soar and ACT-R. Although Street language is parallel by nature, there are synchronisation techniques to tackle race conditions, and make it able to solve deterministic problems.

A conventional computer, with its sequential execution path and centralised memory, is not a good match with the computational requirements of cognitive computing. It would be better served by hardware that supports fine-grained parallelism and which, like the brain, distributes memory and computation. The Street Processor is the hardware designed to fill the gap. It decentralises its memory over an array of alpha memories associated with individual productors. Each productor asynchronously conducts matching over working memory changes and corresponding production rules. The alpha memories are realised using RCAMs that allow parallel search operations to find new rule instantiations.

The Street Processor architecture has some distinguishing features that make it suitable for running advanced cognitive agents. It introduces the concept of a sleep period, during which the placement of the productors is refined to improve overall traffic, latency and power consumption. In addition, a Street Processor has provision for alpha memories to grow over multiple productors using the concept of Big Productors. This allows a Street Processor to run for an extended period of time without memory overflow.

The Street Processor, which contains thousands of homogeneous processing elements, requires an efficient interconnect platform to ensure simultaneous data exchange. The on-chip communication of such processors is discussed in next chapters.

Chapter 4

Interconnect Platform of SoCs with Homogeneous PEs

AN interconnect platform based on network on chip is discussed in this chapter for a system on chip containing many homogeneous processing elements. The NoC architecture is designed to be very simple in order to optimise resource requirements as it is targeted to be used in a system having a large number of simple PEs. This chapter describes various components and design parameters of NoC architectures, including topologies, routing schemes and flow control techniques. This chapter also presents the realisation of a NoC router. This sets the foundation of NoC mapping problem, which is one of the most vital design considerations of NoC design addressed in later chapters.

4.1 Introduction

The scaling of microelectronic technologies has led to an exponential growth of available processing resources on a single chip. However, the performance of SoCs is ultimately limited by the efficiency of interconnection between the PEs, so in many cases the architecture is communication-limited rather that computation-limited. The growing complexity of SoCs requires highly scalable communication infrastructures. The conventional on-chip communication techniques based on point-to-point links and buses cannot provide this scalability. This has inspired researchers to shift to a new paradigm of on-chip communication in the form of networks on chip. In NoC architectures, the network of routers that are associated with PEs, supports simultaneous communication between PEs over multiple links. This can increase the aggregated performance of SoCs.

A number of notable NoC architectures have been reported in the literature. SPIN (Guerrier and Greiner 2000), NOSTRUM (Kumar *et al.* 2002), SoCBUS (Wiklund and Liu 2003), Æthereal (Rijpkema *et al.* 2003b) and Xpipes (Dall'Osso *et al.* 2012) are prominent among them. Each of these architectures has a different router structure, switch and interface design, routing scheme and flow control techniques. However, most of the state of the art architectures are packet-switched and involve regular topologies. Regular topologies are especially suitable for homogeneous SoCs because homogeneous processor architectures tend to exhibit symmetry and regularity in their processing and communication. An example of a regular topology is a mesh, which has a simple layout and switch architecture independent of the network size.

This chapter describes the NoC architectures used in homogeneous SoCs like the Street Processor, which has been discussed in Chapter 3. First, the limitations of traditional on-chip communication methods that led to the development of NoCs are discussed in Section 4.2. Section 4.3 explains different important aspects of NoCs including topologies, routing schemes and flow control techniques. Section 4.4 briefly illustrates the micro-architecture design of a NoC router. Finally, in light of these discussion, Section 4.5 summarises design parameters of the NoC architecture of a Street Processor.

4.2 Limitations of conventional on-chip communication

Small SoCs often use either buses or ad hoc dedicated links as the medium for on-chip communication. With point-to-point links, data travels on dedicated wires directly connecting two end-point PEs, so they potentially yield ideal performance with small number of PEs. However, when the number of on-chip components increases, this scheme requires a huge amount of wiring to directly connect every component to every other. Moreover the utilisation of this wiring is very poor, often less than 10% average wire usage at a time (Dally and Towles 2001). Consequently, the poor scalability due to considerable area overhead is a prohibitive drawback of dedicated links. In addition, the dedicated wires need special attention to manage power, signal integrity and performance issues.

In SoCs with traditional buses, all PEs share the same physical wires and the same limited bandwidth. The serialisation of bus access requests allows only one master at a time to access the buses. This limits scalability and results in significant performance degradation for complex SoCs. Moreover, the addition of new PEs to shared buses increases associated load capacitance, ending up in more energy consumed per bus transaction. The power required to drive long buses with many PEs can be prohibitive (Weste and Harris 2010). In addition, a centralised arbiter of bus-based interconnect adds arbitration latency as the number of PEs increases. As a consequence, the overall performance of a bus-based SoC does not scale with the number of PEs but rather degrades significantly as bus traffic quickly reaches its saturation.

Earlier research (Benini and Micheli 2002, Dally and Towles 2001) pointed out the need for more scalable architectures for on-chip communication, and therefore to progressively replace point-to-point connections and shared buses with NoCs. NoCs provide a much better performance scalability than buses. There is less contention for access to the network infrastructure, since multiple transactions originating from multiple PEs can be handled at the same time. This results in a more efficient network resource utilisation. Thus, for SoCs with a large number of PEs, NoC architectures promise to be the most efficient interconnect solution.

4.3 NoC architecture

The PEs and routers, together with the physical links between routers comprise a NoC architecture. The design of a NoC architecture has several important characteristics that affect its performance: topology, routing, flow control and virtual channels. The next section describes the physical components of NoCs followed by the descriptions of these design considerations.

4.3.1 Physical components

NoC structures can be broken down into four major physical building blocks: PEs, interfaces, routers and links. Figure 4.1 depicts an example of a NoC structure and its major components.

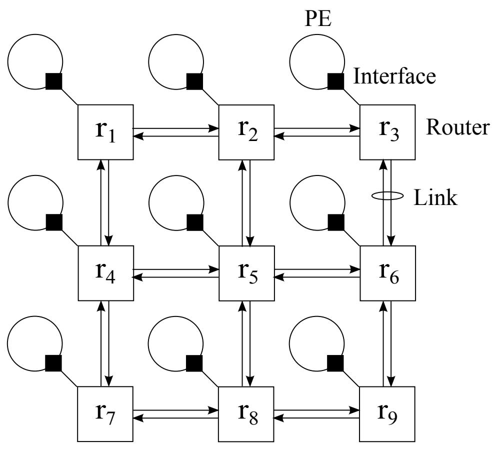


Figure 4.1. A NoC structure and its major components.

Processing elements produce and consume data (*tokens* in the case of the Street Processor) communicated over NoCs. In general, a PE contains a processing core, memory module and possibly other intellectual property (IP) blocks. In the Street Processor, each PE contains a simple and identical productor (described in Section 3.3).

Interfaces couple PEs to the communication infrastructure provided by the NoC. They implement an interface between network protocol and the protocols supported by PEs. Interfaces handle the flow control, encapsulating the data generated by PEs for the routing strategy of the network. The data is broken into *packets* which contain information about their sources and destinations along with payloads.

Routers handle network packets according to defined routing and flow control protocols within the network. They forward packets to adjacent routers via the links in accordance with routing information. Router micro-architecture is discussed in Section 4.4.

Links provide the interconnection between routers in the network. They may consist of one or more logical or physical connections. They usually have two symmetric unidirectional data-paths each with additional control wires both in forward and backward direction. In some NoC designs, control wires enable multiplexing of multiple logical links onto a single physical connection.

4.3.2 Topology

A NoC is functionally composed of links and routers that provide the interconnection mechanism for PEs to transmit and receive packets. The network topology determines the physical layout and connections between routers and links in the network. An important design decision for NoCs concerns topology selection. Topologies can be categorised into regular and irregular classes. Customised domain-specific irregular topologies may offer more power efficiency and deliver better performance to NoCs with specialised heterogeneous PEs (Ishiwata *et al.* 2003, Yamauchi *et al.* 2002). However, for NoCs with homogeneous PEs, regular topologies such as mesh are considered more suitable because of their modularity and regularity (Dally and Towles 2001, Guerrier and Greiner 2000, Kumar *et al.* 2002). These characteristics facilitate re-usability and interoperability of the modules. Moreover, if the network structure is regular and designed beforehand, its electrical parameters can be controlled and optimised very well. Figure 4.2 shows some regular NoC topologies.

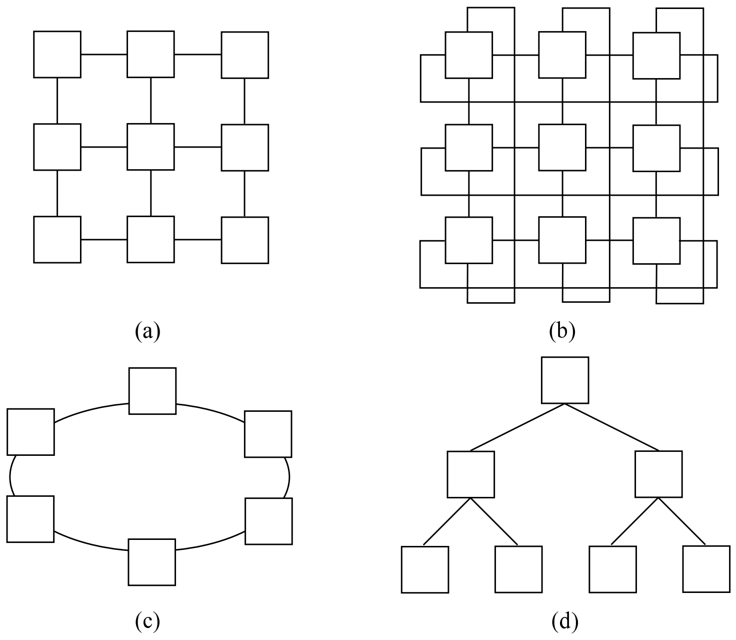


Figure 4.2. Regular NoC topologies. (a) Mesh, (b) Torus, (c) Ring, (d) Tree

Each topology is characterised by several properties, which are briefly discussed here.

Degree: The degree is the number of links connected to each router. Degree determines the number of possible alternate paths out of a router. It also gives an indication of the cost of NoC routers. Routers with higher degrees require more interfaces, which increases implementation complexity. A topology might have a uniform degree, for example, in Figure 4.2 the torus and ring topologies have uniform degrees of 4 and 2 respectively. On the other hand, mesh and tree topologies do not have uniform degrees because the routers at the edges of the topologies have fewer neighbours than the internal routers.

Number of hops: A topology also determines the number of hops (routers) a packet would traverse to reach its destination. This influences network latency significantly. As energy is required to traverse routers and links, the hop count in a topology directly affects network energy consumption.

Path diversity: Furthermore, a topology dictates the path diversity between routers, affecting how well the network can spread out its traffic and hence support bandwidth requirements. This also determines whether the routing algorithm has the flexibility to route packets around faults in the network.

4.3.3 Routing schemes

When a PE wishes to transmit data to another PE it specifies the address or identifier of the destination PE and passes the data into the network through its interface to the attached router. The path that the data takes through the NoC to the destination PE is determined by the routing algorithm. A packet can be transmitted from a source to a destination router using more than one possible path through the network. Two important characteristics of a viable routing algorithm are its livelock and deadlock freedom. A *livelock* occurs when a routing technique never leads a packet to its destination. They can happen in the case of adaptive routings, if the packet is continuously routed in non-productive directions (Glass and Ni 1992). A *deadlock* happens when a packet cannot proceed because some resource that it requires is occupied by another packet and never released. This creates a permanent inability to proceed due to cyclic dependency, unless some corrective measures are taken. Figure 4.3 shows four deadlocked packets waiting in each router for links that are currently held by other packets, preventing any packet from making forward progress. The packet entering router r_1 from the south wants to leave through the east, but another packet is holding that link while waiting at router r_2 to leave via the south output interface, which is again held by another packet that is waiting at router r_4 to leave via the west interface and so on.

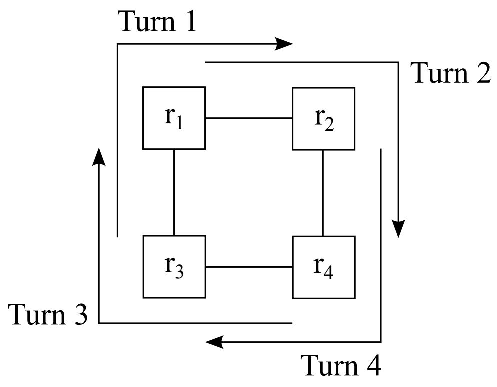


Figure 4.3. A deadlock situation. Turn 1: packet entering r_1 from the south wants to leave through the east, Turn 2: packet entering r_2 from the west wants to leave through the south, and so on.

Many routing algorithms have been proposed (Bjerregaard and Mahadevan 2006, Agarwal *et al.* 2009) in the literature. Deterministic routing is the most commonly used technique among them. In deterministic routing, the routing paths are completely determined by the addresses of sources and destinations (Dally and Towles 2004). As a result, all deterministic routing algorithms are livelock free. Dimension ordered routing is a *minimal* deterministic routing algorithm because it selects paths with the smallest number of hops between the source and destination (Dally and Towles 2004). In this routing, all packets from a source to a destination traverse the same path on a 'per dimension' basis, reaching the ordinate matching its destination before switching to the next dimension. For a 2D mesh, the packet is first routed through X (horizontal, or east-west) axis and then through Y (vertical or north-south) axis until it reaches the destination. For this reason, this routing is also referred as XY routing. Alternatively, if the packets traverse through the Y axis first and then through the X axis, it is called YX routing. XY routing is deadlock free because it allows packets travelling east and west to turn north or south, but prohibits packets travelling north and south from turning east or west. For the same reason, YX is also deadlock free. As illustrated in Figure 4.4, since two of the turns are not permitted the cycles in Figure 4.3 are not possible and hence a deadlock is avoided.

A more sophisticated routing algorithm is adaptive routing, in which the path depends on the network traffic situation. Adaptive routing algorithms often rely on local router information such as queue occupancy and queuing delay to handle congestion and

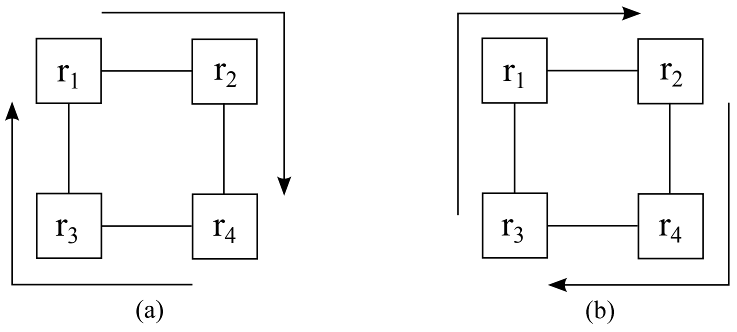


Figure 4.4. Permitted turns in dimension order routing. (a) XY routing turns, (b) YX routing turns

select links (Linder and Harden 1991, Dally and Aoki 1993). With a fully adaptive routing algorithm, livelock and deadlock are crucial problems which are handled by limiting adaptability of the system (Chien and Kim 1995) and by special flow control techniques (Duato 1995). Another problem with adaptive routing is that different packets of the same data (token) can arrive at their destination in an order different from that in which they were sent, which requires special mechanisms and large buffering space to re-order the packets.

Compared to adaptive routing, deterministic dimension order routing is very simple for implementation and especially suitable for SoCs with a large number of PEs. This routing algorithm requires less buffering space since no ordering is required for received packets (Bhattacharyya *et al.* 2003). Furthermore, the predictable traffic of deterministic routing allows the designer to mitigate latency and congestion problems by carefully placing the PEs into the NoC structure.

4.3.4 Routing implementation

Routing algorithms can be implemented using lookup tables at either the source routers or at each hop along the route. These are known as *source routing* and *node table based routing* respectively.

Source routing: In source routing, the route is embedded in the header of the packet at the source. For instance, in Figure 4.1 at page 60, the routing from r_1 to r_9 using XY routing can be encoded as $E \rightarrow E \rightarrow S \rightarrow S$. At each hop, the router reads the most immediate route information from the packet header and sends the packet towards the specified output link. The disadvantages of source routing include the bit overheads required to store the routing table for the entire network at the network interface of each source and to store the entire routing path in each packet. These paths can grow large depending on network size.

Node table based routing: Rather than providing complete routing information in a packet, it can be stored in intermediate routers. This approach significantly reduces the total storage required to hold routing tables because each router stores only direction of the next hop towards each destination rather than the entire path. When multiple paths are available for each destination, this approach supports some sort of adaptability. By updating the routing tables based on the current traffic situation, node table based routing can handle faults and congestion in the network. Table 4.1 demonstrates the routing table with XY routing on the network in Figure 4.1. Each row of the table indicates the next hop directions to reach the corresponding destinations.

Table 4.1. Node table based routing of a 3×3 mesh using XY routing.

Source	**Destination**								
	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9
r_1	-	E	E	S	E	E	S	E	E
r_2	W	-	E	W	S	E	W	S	E
r_3	W	W	-	W	W	S	W	W	S
r_4	N	E	E	-	E	E	S	E	E
r_5	W	N	E	W	-	E	W	S	E
r_6	W	W	N	W	W	-	W	W	S
r_7	N	E	E	N	E	E	-	E	E
r_8	W	N	E	W	N	E	W	-	E
r_9	W	W	N	W	W	N	W	W	-

4.3.5 Flow control techniques

A NoC consists of many links and buffers. Flow control deals with the allocation of links and buffers to a packet as it travels along a path through the network. Whether a deadlocked packet will be dropped, blocked in place, buffered, or re-routed through another link depends on the flow control policy. A good flow control policy avoids link congestion and packet loss while reducing network latency.

Flow control techniques can be classified into two categories: *bufferless* and *buffered*. Bufferless techniques like circuit switching (Lines 2004) and time division multiplexing (Richardson *et al.* 2006), form a path from source to destination by reserving links (not buffers) exchanging by setup and acknowledge messages prior to sending packets. Intermediate routers may buffer the packets temporarily but they should be forwarded in the next cycle. Although latency and buffer requirements are reduced in this method of flow control, it suffers from poor bandwidth utilisation because multiple links will be reserved for the transition of one placket and will not be available for other packets. The links remain idle between setup and the actual packet transfer while other packets seeking to use those resources are blocked. In the buffered flow control technique, the router can store the packet until the required link along the path becomes available. Store and forward (Dally and Towles 2004), virtual cut through (Kermani and Kleinrock 1979) and wormhole routing (Dally and Seitz 1987) are some commonly used buffered flow control techniques. Wormhole routing has been used extensively in high performance parallel computers for its low buffering requirements, which in turn helps routers to meet tight area and power constraints (Jetly 2013).

In wormhole routing, packets are broken down into flow control digits (*flits*), and the flits are routed over the network in a pipelined fashion resulting in a significantly reduced latency. Flits are further broken down into *phits*, which are the smallest physical units that can be transported over a link in a single cycle. Flits can be categorised in three types. The *head flit* of the packet contains routing information and is used to direct the packet to destination. This is followed by payload composed of *body flits* containing data and a *tail flit* indicating the end of a flit sequence. Figure 4.5 depicts the decomposition of transmitted data in wormhole routing. Here VCID is the identity number of the virtual channel (which will be discussed shortly) that the flit occupies.

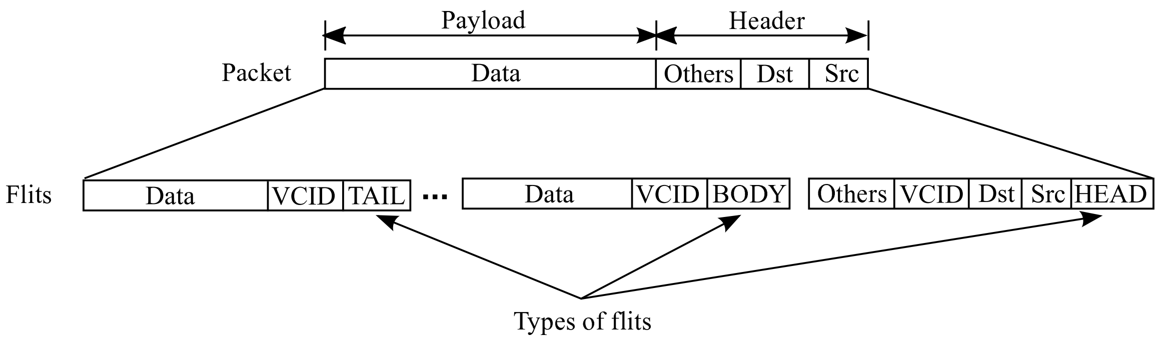


Figure 4.5. Data decomposition in wormhole routing.

In wormhole routing, when the head flit is blocked due to congestion, the trailing flits of the packet wait at their current routers. As a result, the routers do not need large buffers to store the whole packet, rather small buffers to store several flits are sufficient. As the packet size does not depend on the available buffer size, it gives more flexibility to define the packet size minimising control overhead in wormhole routing. Because of limited buffering requirements and reduced latency (Dally and Seitz 1987), wormhole routing is believed to be the most appropriate flow control technique for systems with a large number of PEs.

4.3.6 Virtual channels

When a packet at a router's input interface is blocked, it stalls subsequent packets that are in the queue behind it, even if resources are available for the stalled packets. The use of virtual channels is a flow control technique that reduces this *head-of-line blocking problem* (Dally 1992). It splits a physical link into a number of virtual channels and associates multiple virtual channels with each router interface. Virtual channels arbitrate for physical link bandwidth on a 'per cycle' basis. When a packet holding a virtual channel becomes blocked, other packets can still traverse the physical link through other virtual channels. Thus virtual channels increase the utilisation of the physical links and extend overall network throughput.

Figure 4.6 illustrates an example of virtual channels. Packet p_1 arrives at the west interface of router r_1 and with an intention to reach r_4, occupies the link between r_1 and r_2. Now if another packet p_2 arrives later in the north interface of r_1 with destination to r_3, in the absence of virtual channels, p_2 will be blocked until p_1 releases the link. However, with virtual channels, the physical link will be multiplexed and both packets will be able to proceed without blocking each other.

4.4 Router micro-architecture

Figure 4.7 illustrates the micro-architecture of a NoC router for wormhole routing with n interfaces, each having v virtual channels (Dally and Towles 2004). The major components of a router are input buffers, crossbar switch, route computation logic, switch allocator and virtual channel allocator. The first two components provide the data-path of the router, and are involved in storage and movement of a packet's payload.

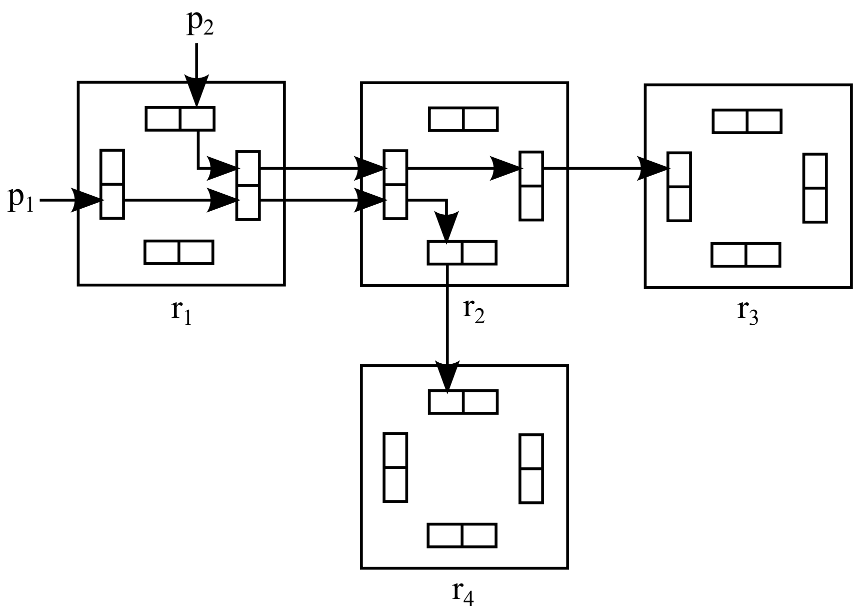


Figure 4.6. Example of virtual channel flow control.

The others are considered in as the *control unit*, which coordinates the movement of packets through the resources of the data-path. The number of interfaces of a router depends on the number of PEs attached to it. It is a common practice to attach one PE to a router. However, for SoCs that have a large number of PEs, it is advantageous to associate multiple PEs to a single router in order to reduce the number of routers and overall cost of the NoC architecture.

The physical components of a router are discussed here.

Input buffers are first-in first-out (FIFO). They hold the incoming flits until they are released to the output interfaces. Buffer cells can be implemented using static latches or SRAMs depending on the buffer size and access timing requirements. For NoC routers that have several interfaces to support multiple PEs per router, the buffers are normally implemented using static latches, typically in the size of a small number of flits each. These can be readily synthesised without requiring memory generators.

Crossbar switch is responsible for transferring flits from input to output interfaces. The router uses this crossbar network with full connectivity, enabling multiple and simultaneous flit transfers. Hence the size of the crossbar is $vn \times n$. This is built using multiplexers that receive *select* signals from the switch allocator to set up corresponding connections between input buffers and output interfaces.

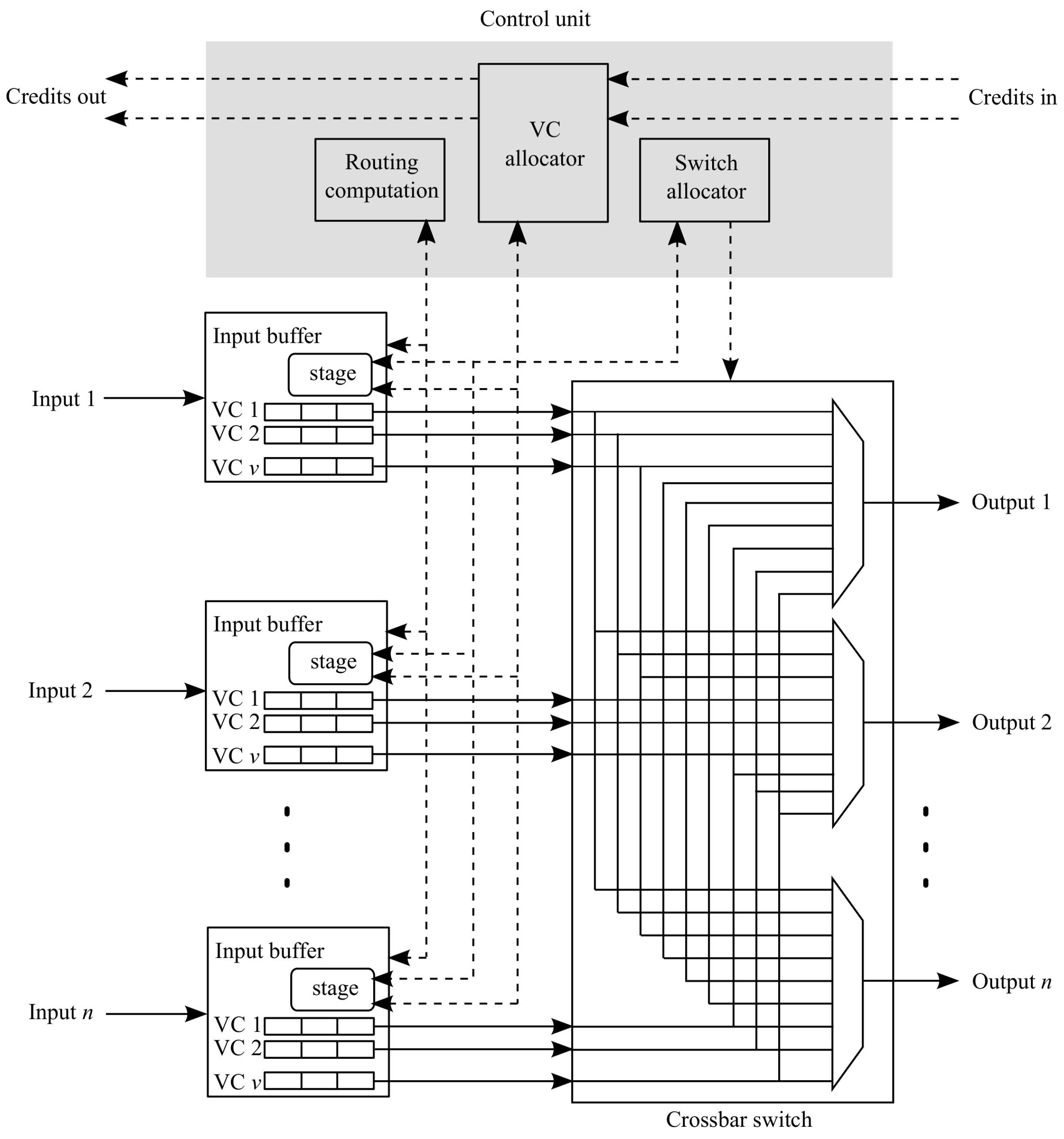


Figure 4.7. Router micro-architecture with n interfaces, each having v virtual channels. The solid and dotted lines indicate the data-paths and the control signals respectively.

Control unit comprises a routing computation block, a virtual channel allocator and a switch allocator. The routing computation logic looks up the routing table for the output interface to which the the flits of a packet will be forwarded. Once the output interface is determined, the flits are allocated a virtual channel by the virtual channel allocator. Each flit of the packet is then forwarded over the virtual channel by assigning a time slot on the switch using the switch allocator.

A switch allocator provides the necessary signals to the crossbar switch to move flits from the input buffer to the output interface.

The router employs credit-based link level flow control to coordinate flit delivery between routers. Credits keep track of the number of buffers available at the next hop, by sending a credit to the previous hop when a buffer is vacated, and incrementing the credit count at the previous hop upon receiving the credit. When a flit departs the current router, the current router decrements the credit count for the appropriate downstream buffer.

The router works in a pipeline at the flit level. Figure 4.8 illustrates the logical pipeline stages of the router. When a head flit arrives at the input interface of a router, it is first decoded and stored at the input buffer in the buffer write (BW) pipeline stage. In the next stage, the routing logic performs route computation (RC) based on the information in the head flit, to determine the output interface to which the packet will be forwarded. The head flit then enters the virtual channel allocation (VA) stage to arbitrate for a virtual channel at the allocated output interface. Upon successful allocation of a virtual channel, the head flit proceeds to the switch allocation (SA) stage where it arbitrates for the switch. On achieving switch arbitration, the flit is read from the input buffer and traversed over the crossbar switch to the output interface in the switch traversal (ST) stage. Finally, the flit is passed to the next router in the link traversal (LT) stage. Body and tail flits follow a similar pipeline except that they do not go through RC and VA stages, instead inheriting the route and the virtual channel allocated to the head flit. The tail flit, on leaving the router, deallocates the virtual channel reserved by the head flit.

Head flit	BW	RC	VA	SA	ST	LT			
Body flit		BW			SA	ST	LT		
Body flit			BW			SA	ST	LT	
Tail flit				BW			SA	ST	LT

Figure 4.8. Pipeline stages of a NoC router.

4.5 NoC of the Street Processor

A Street Processor will typically thousands of fine-grained homogeneous PEs executing in parallel. Unlike SoC implementations of multimedia and image processing algorithms, which usually contain complex processing nodes e.g. CPUs, DSPs and RAMs, the PEs of a Street Processor are relatively simple. Thus, a regular and generic tile-based NoC architecture is expected to be sufficient to meet its communication requirements. Each tile contains a router, and one or more PEs. A regular mesh topology fits well for this kind of large homogeneous system. A dimension order XY routing is used in this NoC design because it is immune to deadlock and livelock. Furthermore, this routing scheme requires less buffering space and is very simple to implement.

Since this interconnect architecture is designed to support a large number of PEs, buffer space optimisation is very crucial. To this end, node table based routing is proposed, as it only requires space to store routing information for the next hop towards the destination, rather than the entire path. The NoC of the Street Processor uses wormhole routing as a flow control policy, for its low buffering requirements and reduced latency. The optimal size of the buffers is a function of the technology in which the architecture is implemented (FPGA, full custom, etc.) and is not addressed in detail in this work.

As the routers are expensive components of a NoC architecture, each of the routers of the Street Processor is connected to multiple PEs, instead of one single PE. The number of PEs associated to individual routers depends on the traffic between PEs, which is discussed in Chapter 6. As the highly communicating PEs are placed under the same router, the traffic is handled internally. This, in turn, reduces the network traffic. To implement multiple interfaces with reduced buffering resources, static FIFO buffers that can store a small number of flits are used in each interface of the routers. In wormhole routing, since the body and tail flits follow the head flit, and a single buffer does not need to store the whole packet, the flits of a packet can span over multiple buffers trailing behind the head flit.

4.6 Conclusion

In this chapter, an interconnect platform suitable for SoCs with a large number of homogeneous PEs has been described. Point-to-point links and shared buses cannot provide enough scalability to support large and complex SoCs like the Street Processor.

Rather, they limit the overall performance of the system if the power and traffic are not managed very carefully. NoC architectures provide a more scalable and efficient alternative communication for such SoCs by allowing multiple simultaneous transactions over the network (Dally and Towles 2001).

A regular mesh topology is advocated as suitable for NoCs supporting a large number of PEs because of their modularity and re-usability. A deterministic dimension order routing is preferred over adaptive routing for its ease of implementation and lower buffering requirements. To further optimise the buffering space, the NoC design uses wormhole routing, which requires small buffers in NoC routers, good enough to store several flits of a packet. As the packet size and the buffer size are independent to each other, it provides more flexibility to define the packet size. This chapter also discusses the micro-architecture of a NoC router supporting these aspects.

Although this chapter describes many important parameters and implementation aspects of a NoC architecture, it does not discuss the placement of PEs onto NoC structures. This is a very complex and delicate problem of NoC design. The performance of a system varies significantly depending on NoC mapping (Hu and Marculescu 2003b, Lahiri *et al.* 2001). Chapters 5 and 6 address this crucial aspect of NoC design in detail.

Chapter 5

Mapping of Homogeneous PEs to NoC Routers

THIS chapter describes the mapping of processing elements to network on chip routers. We explore two optimisation techniques to solve this mapping problem. The Simulated Annealing is a smart algorithm to solve optimisation problems having multiple local minima. It probabilistically allows hill-climbing to escape from the trap of local minima. On the other hand, The Branch and Bound is a systematic approach, which limits its search spaces using heuristics. We show that Branch and Bound can be implemented in Street language to build a self-configurable agent, which can re-arrange its PE placements for improved performance. This agent, in turn, is used as a test case to evaluate the performance of the mapping techniques.

5.1 Introduction

Research on NoC design can be broadly classified into four categories (Marculescu *et al.* 2009, Agarwal *et al.* 2009). The first focuses on the choice of communication infrastructure, such as network topology, router architecture, buffer optimisation, link design and clocking (Murali and De Micheli 2004, Rijpkema *et al.* 2003a, Wolkotte *et al.* 2005). The second deals with the communication paradigm including routing techniques, switching methods, congestion control, power and thermal management (Glass and Ni 1992, Siebenborn *et al.* 2004, Hu and Marculescu 2004, Shang *et al.* 2006). The third dimension involves designing an evaluation framework for NoCs to obtain a good understanding of achievable throughput, latency, and bandwidth of the network (Carloni *et al.* 2001, Lin and Pileggi 2002). These three fields of research build the communication infrastructure and paradigm of NoCs. The fourth important direction of NoC research is to optimise the association of PEs with routers. This plays a very significant role in determining the performance of the overall system, as it directly influences communication time, required link bandwidth and power consumption.

A number of NoC mapping techniques proposed in the literature have been reviewed in Chapter 2. Mapping techniques can be either static or dynamic. In static mapping, the placement of PEs on routers is determined before the system starts, and it remains unchanged during execution. On the other hand, if the mapping changes during execution, it is a dynamic mapping. This is typical for NoCs that are fault-tolerant and adaptive by nature. The cost of dynamic mapping often outweighs its benefits (Carvalho *et al.* 2009). Static mapping techniques can be categorised into deterministic and heuristic approaches. A deterministic mapping attempts to constructively enumerate the solution space while a heuristic mapping does not. A heuristic is a local search method relying on a neighbourhood function to search near-optimal solutions (Aarts and Lenstra 1997). A deterministic approach fundamentally involves Branch and Bound or Back-tracking algorithms, but Back-tracking has been criticised for high computation time (Gendron and Crainic 1994). On the other hand, among the heuristic approaches such as Simulated Annealing, Tabu search, genetic algorithms and neural networks, Simulated Annealing provides a good approximation to the global minimum of a given function in a large space (Aarts and Lenstra 1997).

In Chapter 4, a NoC communication platform has been presented for systems on chip having multiple homogeneous PEs. This chapter describes the process of PE mapping onto the network structure. To this end, mapping techniques based on Simulated Annealing, and Branch and Bound are considered from implementation perspective in this chapter because of their advantages over other candidates of the same category. The mapping algorithms are applied to graphs constructed from a recent history of traffic between the PEs of the Street Processor, during the sleep period (discussed in Section 3.7). The chapter starts with a discussion on the aspects to be considered during NoC mapping in Section 5.2 and the definition of the mapping problem in Section 5.3. The mapping techniques are discussed in Sections 5.4 and 5.5. The *Street* implementation of the Branch and Bound mapping is discussed in Section 5.6. This also acts as a test case to evaluate the performance of the mapping techniques. In Section 5.7, these mapping techniques are compared based on the test case and a range of network sizes.

5.2 Mapping considerations

An algorithm for mapping PEs to NoC routers must be aware of the *network topology*. Topology may be defined as a physical layout and connections between routers and channels in the network. The effect of a topology on overall network cost-performance is profound. A topology determines the number of routers that a packet must traverse, thus influencing network latency significantly. As traversing routers and links incurs energy, a network topology also directly affects network energy consumption. Furthermore, the topology dictates the total number of alternate paths between PEs, affecting how well the network can spread out traffic and hence support bandwidth requirements.

Secondly, it is also crucial to set *mapping constraints* carefully. A mapping constraint can be defined as a restriction derived from the requirements of the system and the characteristics of the NoC architecture, imposed when associating PEs to network structure. It is very likely that not every mapping is feasible in all cases. Any mapping constraints are likely to limit the size of the search space and speed up the mapping algorithm. The bandwidth requirement is an example of a mapping constraint.

Thirdly, a mapping algorithm explores the search space of possible mappings to find the best mapping. In order to determine the best mapping, at least one *optimisation goal* is required. Network throughput or latency, communication energy, power consumption and computation time are some examples of optimisation parameters. Thus, a mapping algorithm searches for the best mappings by attempting to optimise one or more such parameters.

Finally, the mapping problem is closely related to *routing*. Any routing algorithm may be applied after the mapping has been done. However, if the routing method is not defined beforehand, it is possible that the mapping is sub-optimal because it has failed to take into account the true costs of packet routing. A mapping algorithm should therefore consider the routing paths for the mapped PEs as well. The routing function can be deterministic or adaptive. Also, it should provide freedom from deadlock and livelock as previously discussed.

In this chapter, we consider the mapping problem for the NoC of the Street Processor, as designed in Section 4.5. Recall that this NoC is a regular, tile-based network architecture, which considers a mesh topology. During a sleep period, the system updates its PE placements, depending on the traffic statistics, but this is not considered to be a dynamic mapping because the mapping can only be changed while the system is inactive. The mapping constraints and optimisation goals can be described in the form of an energy cost function, which will be discussed in Section 5.3. As explained in Section 4.3.3, the deterministic dimension order routing is used because of its livelock and deadlock freedom.

5.3 Mapping constraint and goal

This section formulates the PE-to-router mapping problem by defining the optimisation goal in terms of an energy cost function.

5.3.1 Energy cost function

In (Walter *et al.* 2009), the energy cost is defined in terms of total bandwidth delivered by the links used. Let us assume that in a NoC structure, there are N_R routers and equal number of PEs to be mapped to those routers. If in a particular mapping $M \epsilon \mathbb{M}$ where $\mathbb{M}$ is the set of all possible mappings, PE $p_i \epsilon \mathbb{P}$ is mapped to router $r_i \epsilon \mathbb{R}$, where $\mathbb{P}$ and $\mathbb{R}$ are sets of PEs and routers respectively, and $l_{i,j} \epsilon \mathbb{L}$ is a physical link between router r_i and r_j, the energy cost function to minimise the total bandwidth required is expressed by (5.1)

$$Cost(M) = \sum_{1 \le i,j \le N_R} comm(p_i, p_j) \times d_{i,j} \tag{5.1}$$

Here, $d_{i,j}$ is the shortest Manhattan distance between routers r_i and r_j, and $comm(p_i, p_j)$ is the average data transfer rate between r_i and r_j observed over some defined time interval (in bits/second). In order to demonstrate how the NoC mapping affects the overall cost, let us consider the two example mapping instances shown in Figure 5.1. They consist of nine PEs placed onto a 2D mesh NoC. In each case, p_2 communicates 30 bits/second to p_6 and p_4 communicates 100 bits/second to p_3 based on the action elements of corresponding production rules. For the sake of simplicity, the communication between other PEs are ignored here.

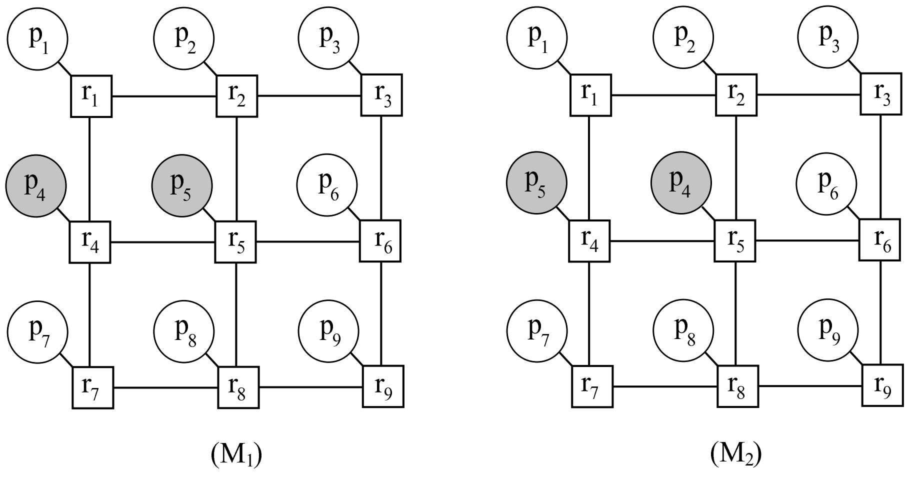


Figure 5.1. Examples of two mappings M_1 and M_2.

In mapping M_1, p_2 is mapped to r_2, p_6 to r_6, p_4 to r_4 and p_3 to r_3. In mapping M_2 we swap p_4 to r_5 and p_5 to r_4. The energy costs of these two arrangements are

$$Cost(M_1) = comm(p_2, p_6) \times d_{2,6} + comm(p_4, p_3) \times d_{4,3} = 30 \times 2 + 100 \times 3 = 360 bits/second \tag{5.2}$$

$$Cost(M_2) = comm(p_2, p_6) \times d_{2,6} + comm(p_4, p_3) \times d_{4,3} = 30 \times 2 + 100 \times 2 = 260 bits/second \tag{5.3}$$

It can be noticed that because of difference in placement of p_4, mapping M_2 provides lower cost than M_1.

In (Ye *et al.* 2002), the authors represented this cost function using an energy model. According to this, the energy consumed in sending one bit of data from r_i to r_j is calculated as

$$E_{r_i,r_j} = E_s \times \left(d_{i,j} + 1\right) + E_l \times d_{i,j} + E_b \times \left(d_{i,j} + 1\right) \tag{5.4}$$

where E_s, E_l and E_b are the energy consumed at each switch, link and buffer along the path from router r_i to r_j, measured in Joules/bit. So, if n bits of data are transmitted between r_i and r_j, the energy cost becomes

$$E_{r_i,r_j} = n \times \left\{E_s \times \left(d_{i,j}+1\right) + E_l \times d_{i,j} + E_b \times \left(d_{i,j}+1\right)\right\} \tag{5.5}$$

The total energy cost of a particular mapping, M, thus can be obtained from (5.6), where N_R is the number of routers in the network structure.

$$Cost(M) = \sum_{1 \leq i,j \leq N_R} E_{r_i,r_j} \tag{5.6}$$

In (5.1) and (5.6), E_s, E_l and E_b are constants for a given NoC architecture and implementation technology. Hence, the energy cost effectively depends on the distance $d_{i,j}$ between the mapped PEs which makes NoC mapping very crucial.

5.3.2 Problem definition

To define the NoC mapping problem it is necessary to define some important terms first.

Dependency graph (DG): $DG = (\mathbb{P}, \mathbb{D}, W_{\mathbb{D}})$, is a directed graph, which quantifies the dependencies between PEs. Here, $\mathbb{P}$ is the set of PEs, and $\mathbb{D}$ is the set of dependencies between them represented by directed edges, each of which connects an ordered pair of PEs. The dependency from p_i to p_j, where $p_i, p_j \epsilon \mathbb{P}$, means that in order to instantiate or execute p_j, necessary information is required to arrive from p_i. It is represented by a directed edge $(p_i, p_j) \epsilon \mathbb{D}$, which is associated with the communication volume from p_i to p_j measured over a defined interval. The mapping between $\mathbb{D}$ and the set of communication volumes $\mathbb{C}$ is represented by the function $W_{\mathbb{D}} : \mathbb{D} \rightarrow \mathbb{C}$.

Traffic graph (TG): $TG = (\mathbb{P}, \mathbb{T}, W_{\mathbb{T}})$ is a weighted undirected graph derived from the DG, which represents total traffic flow between the PEs. Here, $\mathbb{P}$ is the set of PEs, $\mathbb{T}$ is the set of edges representing inter-dependencies between the PEs, and $W_{\mathbb{T}} : \mathbb{T} \rightarrow \mathbb{C}'$ represents the function from $\mathbb{T}$ to the set of total traffic $\mathbb{C}'$ between the PEs. Each edge $(p_i, p_j) \epsilon \mathbb{T}$, connecting PEs p_i and p_j, is associated with the total traffic between the two PEs. If in a DG, there are edges $(p_i, p_j) \epsilon \mathbb{D}$ and $(p_j, p_i) \epsilon \mathbb{D}$, between two PEs p_i and p_j, having weights $W_{\mathbb{D}}((p_i, p_j))$ and $W_{\mathbb{D}}((p_j, p_i))$ respectively, then the weight of the traffic edge $(p_i, p_j)\epsilon\mathbb{T}$ in TG is derived to be $W_{\mathbb{D}}((p_i, p_j)) + W_{\mathbb{D}}((p_j, p_i))$.

Architecture graph (AG): $AG = (\mathbb{R}, \mathbb{L}, W_{\mathbb{L}})$, is an undirected graph that describes the NoC structure. Here $\mathbb{R}$ is the set of routers, and $\mathbb{L}$ is the set of physical links. $W_{\mathbb{L}} : \mathbb{L} \rightarrow \mathbb{B}$ is a function mapping $\mathbb{L}$ to the set of link capacities $\mathbb{B}$. In static mapping, the link capacity is considered equal for all the links.

Assuming that the number of PEs is not greater than the number of routers, the objective is to find a mapping function: $M : \mathbb{P} \rightarrow \mathbb{R}$ such that the cost in (5.6) is minimised. In other words, the objective of the mapping problem is to find the best mapping of PEs to routers that minimises overall energy consumption, and thus makes the NoC architecture more power-efficient.

However, mapping of PEs to routers is an instance of an NP-hard optimisation problem, in which the search space of the problem increases factorially with the system size (Garey and Johnson 1979). For NoCs with N_P PEs and N_R routers, where $N_P \leq N_R$, the size of the search space is $\frac{N_P!}{(N_R - N_P)!}$, considering each PE is attached to one router. This means, for a small problem of the mapping 16 PEs to an equal number of routers, there is a search space of $16! = 2.092279 \times 10^{13}$. If each search takes 0.01 millisecond, it will take 6.6 years to complete the entire search. It is therefore important to take a strategic decision about the mapping technique to optimise the performance metrics with an acceptable trade-off between optimality of the mapping and computation time. Hence, the optimisation goal of this work is defined to be: to find the mapping with near-optimum energy cost in a short period of time.

5.4 Simulated Annealing

Simulated Annealing (SA) (Kirkpatrick *et al.* 1983) is an iterative probabilistic algorithm for solving optimisation problems. It includes the notion of hill-climbing which makes this algorithm especially suitable for the problems with a lot of local minima. The idea behind this algorithm comes from metallurgy, where annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to coalesce into a low-energy crystalline state. A typical local search algorithm proceeds by choosing a random initial solution and generating a neighbour from that solution. The neighbouring solution is accepted if it has a lower cost. Such an algorithm has the risk of converging to a local minimum. Although Simulated Annealing algorithm is by itself a local search algorithm, it avoids getting trapped in local minima by occasionally accepting neighbouring solutions with higher cost with some probability. The advantages of this technique are its ease of implementation, its applicability for many combinatorial optimisation problems and the ability to give reasonably good solutions.

5.4.1 Simulated Annealing algorithm

In this work, Simulated Annealing is applied on the traffic graph. It effectively optimises solutions over large state spaces by making iterative improvements. It has a concept of *temperature*, which is initially set very high, and keeps reducing at every step. It starts with a random solution, and searches for better solutions that pass the temperature-dependent acceptance test. Figure 5.2 shows the pseudo-code for this algorithm. As parameters, the algorithm requires the initial temperature T_0 and the maximum number of iterations per temperature, which is equal to the number of routers N_R. It returns the optimised mapping M_{best} as an output. The algorithm has some important concepts and steps that are described in the following subsections.

5.4.2 Annealing schedule

A geometric annealing schedule is used in this mapping technique as recommended by Fouskakis and Draper (2002). The geometric annealing temperature schedule defines the temperature at iteration i as $T_i = T_{i-1} \times 0.9^{\lfloor i/N_R \rfloor}$ in line 22 of Figure 5.2. Here the geometric progression ratio is set to 0.9 following (Kirkpatrick *et al.* 1983). The initial temperature T_0 is set sufficiently high to accept virtually all transitions. But there is no minimum temperature, rather a *stop criterion* is checked to determine when annealing should stop.

5.4.3 Acceptance test

Simulated Annealing probabilistically accepts bad moves to avoid traps of local minima. For this occasional hill-climbing, the Metropolis method (Catoni 1996) is used

```
Require: $TG, T_0, N_R$
 1: $T = T_0$
 2: $M = RandomMapping(TG)$
 3: $C = M.GetMappingCost()$
 4: $done = false$
 5: while $done \neq true$ do
 6:     for $i = 1$ to $N_R$ do
 7:         $M_{new} = RandomSwap(M)$
 8:         $C_{new} = M_{new}.GetMappingCost()$
 9:         $\Delta C = (C_{new} - C)/C$
10:         if $\Delta C < 0$ or $F(\Delta C) \geq random(0, 1)$ then
11:             $M = M_{new}$
12:             $C = C_{new}$
13:         else
14:             $M = RollbackSwap(M_{new})$
15:         end if
16:     end for
17:     if $StopTest(C, C_{prev})$ then
18:         $M_{best} = M$
19:         $C_{best} = C$
20:         $done = true$
21:     else
22:         $T = T \times 0.9^{\lfloor i/N_R \rfloor}$
23:         $C_{prev} = C$
24:     end if
25: end while
26: return $M_{best}$
```

Figure 5.2. Simulated Annealing algorithm for NoC mapping.

in line 10 of Figure 5.2. The acceptance probability function is $F(\Delta C) = exp(-\Delta C/T)$ according to this method, where T is the temperature and ΔC is the cost difference between two consecutive mappings. This equation indicates that a decrease in ΔC gives higher acceptance probability, and an increase of ΔC results in lower acceptance probability. As the number of iterations increases and the temperature decreases, the acceptance probability for a given ΔC decreases. The acceptance probability of the Metropolis criterion is implemented by comparing $F(\Delta C)$ with $random(0, 1)$ which is a random number generated with a uniform distribution between 0 and 1. The intention is to accept all moves producing a decrease in cost and probabilistically accept moves producing an increase in cost, to escape from local minima.

5.4.4 Swapping function

A uniform random swap function $RandomSwap()$ is used to obtain a new mapping M_{new} from the current mapping M. Suppose that, in current mapping M, PE p_i is allocated to r_i, and the new mapping M_{new} is derived by applying the function on M. The function randomly selects a PE p_j from a router $r_j \neq r_i$ and exchanges p_i with p_j. If the acceptance test in line 10 of Figure 5.2 is not passed, the $RollbackSwap(M_{new})$ function in line 14 rollbacks this swap operation.

5.4.5 Stop criterion

The stop criterion at line 17 of Figure 5.2 is checked in every iteration to determine whether the cost has changed negligibly over the last several temperatures or the acceptance test is passed. When a certain number of consecutive iterations result in the same energy cost and ΔC is insignificant, it indicates that the annealing is not giving any better mapping, and hence should be terminated.

The performance of Simulated Annealing to map PEs of a Street Processor is analysed in Section 5.7.

5.5 Branch and Bound

The Branch and Bound (BB) algorithm for NoC mapping was proposed in (Hu and Marculescu 2005) with the purpose of optimising total communication energy while satisfying bandwidth constraints. It is a systematic search algorithm that topologically finds the best mapping by searching the solution in tree branches, and bounding unallowable solutions.

In this mapping technique, a search tree is generated, which represents the solution space. The *root node* corresponds to the state where no PE is yet mapped to any router. Each *internal node* represents a partial mapping, and each *leaf node* is a complete mapping of PEs onto routers. Each node has a mapping cost attached with it. This cost represents the energy necessary for the communication among the PEs already mapped. The cost of a child node cannot be less than the cost of its parent node because the child contains one more PE mapped to a router. Figure 5.3 shows an example search tree of the solution space. For instance, the internal node labelled $r_0xx...x$ represents a partial arrangement where p_0 is mapped to r_0, and other PEs are yet to be mapped. Again, the leaf node $r_0r_{n-1}r_1...r_{n-2}$ represents a complete mapping in which PEs p_0, p_1, p_2, ... p_{n-1} are mapped to routers r_0, r_{n-1}, r_1, ... r_{n-2} respectively.

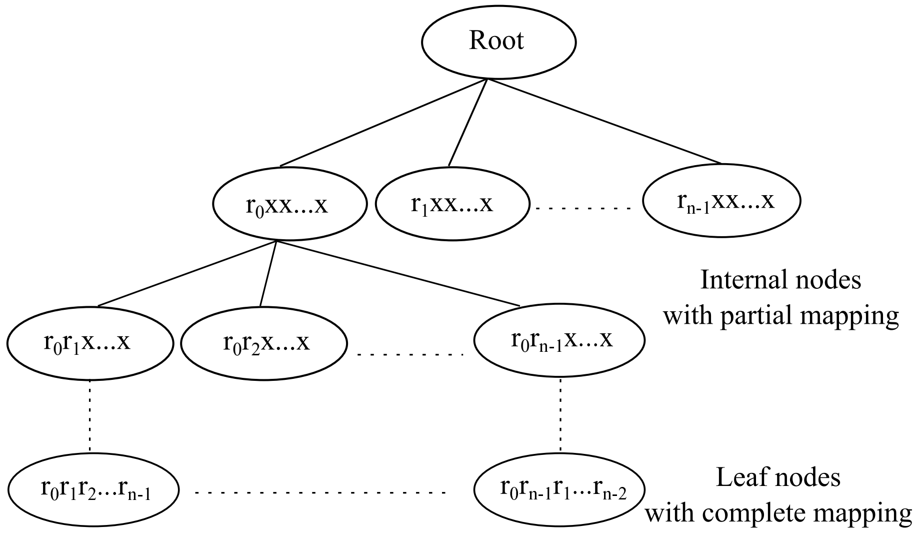


Figure 5.3. An example of Branch and Bound search tree of solution space.

5.5.1 Branch and Bound algorithm

As the name suggests, the Branch and Bound algorithm has two major steps:

Branch: The next unmapped PE of an unexpanded node is enumeratively assigned to unoccupied routers, and the child nodes are generated. For example, the internal node $r_0xx...x$ is expanded to $r_0r_1x...x$, $r_0r_2x...x$, ... $r_0r_{n-1}x...x$ meaning the next unmapped PE p_1 is mapped to routers r_1, r_2, ... r_{n-1} respectively.

Bound: The new child node is checked whether it satisfies mapping constraints, and also if it can generate the best leaf nodes. If not, the node is not expanded further.

The PEs are initially sorted based on their traffic demands that are obtained from the traffic graph. The traffic demand of a PE is calculated by summing up all the traffic volume associated to it. As the PEs with higher traffic demands dominate the overall energy consumption, they are mapped first to the unoccupied routers to generate new child nodes. A node is expanded, if and only if the mapping constraint is met by the PEs already mapped. When a node is found unexpandable, the subsequent nodes derived from it are not capable of providing the best mapping, and thus not explored.

Each of the newly generated child nodes are examined to see if it is possible to generate the best leaf node from it. The algorithm calculates the upper and lower limits of costs of the child nodes to detect candidate optimal nodes.

Upper bound cost of a node is the value that is no less than the minimum cost of its leaf nodes. Thus, UBC is a metric that informs the highest cost of the best mapping that may result by expanding the current node.

Lower bound cost is defined as the lowest cost that its leaf nodes can possibly achieve. LBC indicates the lowest cost of the best mapping that may result from the the current node.

The cost of leaf nodes generated from a node will be between the LBC and UBC of this node. If the cost or LBC of a node is higher than the lowest UBC that is already found so far, it is deleted without any expansion, because it is guranteed that the cost of the leaf nodes are certainly greater than the lowest UBC and the node cannot lead to the best mapping solution. The lowest UBC and highest LBC are updated in every step. All the nodes are traversed this way, and finally the node with minimum cost is accepted as the best mapping node. The speed of this algorithm depends on the computation of UBC and LBC. These are described in the following subsections.

Figure 5.4 shows the pseudo-code of the mapping algorithm. A priority queue is used in this algorithm to store the nodes. It starts with the root node at line 5. Each time a node is inserted into the queue, it is sorted according to energy cost (line 24) so that the node with the lower cost gets higher priority because the node with the lowest cost is more likely to result in leaf node with the minimum cost.

```
Require: $TG$
 1: Sort PEs of $TG$ by traffic demand
 2: $M_{root} = NULL$
 3: $UBC_{min} = \infty$
 4: $C_{best} = \infty$
 5: $Q.Insert(M_{root})$
 6: while $Q.Next() \neq NULL$ do
 7:     $M_{cur} = Q.Next()$
 8:     for all unoccupied router $R_i$ do
 9:         $M_{new} = M_{cur}.GetChildNode(R_i)$
10:         $C_{new} = M_{new}.GetMappingCost()$
11:         if $C_{new} > C_{best}$ or $M_{new}.LBC > UBC_{min}$ then
12:             continue
13:         end if
14:         if $M_{new}.IsLeafNode()$ then
15:             if $C_{new} < C_{best}$ then
16:                 $C_{best} = C_{new}$
17:                 $M_{best} = M_{new}$
18:             end if
19:         else
20:             if $M_{new}.UBC < UBC_{min}$ then
21:                 $UBC_{min} = M_{new}.UBC$
22:             end if
23:             $Q.Insert(M_{new})$
24:             $Q.Sort()$
25:         end if
26:     end for
27: end while
28: return $M_{best}$
```

Figure 5.4. Branch and Bound algorithm for NoC mapping.

5.5.2 Lower bound cost calculation

The LBC of a node is computed as the sum of three components: energy cost between mapped PEs, between unmapped PEs, and between mapped-unmapped PEs as described by (5.7). Here, m and m' subscripts indicate the mapped and unmapped PEs respectively.

$$LBC = Cost_{m,m} + Cost_{m,m'} + Cost_{m',m'} \tag{5.7}$$

The cost between mapped PEs can be calculated exactly. The other two components are heuristically determined. To calculate the cost between unmapped PEs, two unoccupied routers with the minimum distance are determined and the unmapped PEs are considered to be mapped to them, and the cost is calculated. And, to calculate the cost between mapped and unmapped PEs, the unoccupied router with the shortest distance from the occupied routers, is determined and the unmapped PEs are considered to be mapped to it, and the cost is determined.

5.5.3 Upper bound cost calculation

By definition, the cost of any legal child node can be considered as the UBC of the parent node, but a tight UBC is desired. So the minimum UBC is updated in each iteration. A greedy method, which maps the unmapped PEs to unoccupied routers, is used to determine the child node with the smallest cost. A UBC also has three components: cost between originally mapped PEs, between originally mapped and greedily mapped PEs, and between greedily mapped PEs as in (5.8). The subscripts m and g indicate the originally mapped and greedily mapped PEs respectively.

$$UBC = Cost_{m,m} + Cost_{m,g} + Cost_{g,g} \tag{5.8}$$

At each step, the greedy method takes the next unmapped PE with the highest communication demand, and places it on the router which best facilitates the communication of the PE to the originally mapped PEs. If the router is already occupied, then the unoccupied router with smallest Manhattan distance from this router is considered. This step is repeated until all the PEs are mapped. If the child node is illegal, the UBC of the current node is set to be infinitely large, otherwise it is set equal to the cost of the child node.

The performance of UBC computation is thus determined by the speed of the greedy mapping. The greedy method effectively tries to do a quick mapping to search for the best mapping cost that may be achieved starting from the current node of the search tree. The accuracy of the UBC value strongly depends on the accuracy of the greedy mapping.

The Branch and Bound mapping implementation using the Street language is described in the next section with a view to build a self-configurable agent, which can re-arrange the placements of the PEs on a NoC structure to improve its performance.

5.6 Test case: the self-configurable Street agent

Street agents are intended to run for an extended period of time in different environments. Based on varying inputs from the environment, production rules may fire at different rates, and dependency and traffic graphs may vary over time. There is a possibility that after a period of execution the previous mapping of PEs onto the network structure is not providing good performance any more. This might happen because of decrease of traffic loads between adjacent PEs and increase of traffic loads between distant PEs. Hence, if the mapping of PEs is updated periodically based on current traffic, the agent is expected to perform better than before. A feature of the Street engine is that during a sleep period (discussed in Section 3.7), PEs are re-arranged on the network infrastructure without external intervention. This self-configuration provides the following advantages over static configuration:

Adaptability: The self-configurable agent can change its PE mapping to adapt to the changing environment.

Autonomy: The agent does not need any external help for reconfiguration; this provides autonomy to the agent.

Versatility: The PEs can be mapped in different ways making the same agent to improve its performance for input changes over time.

With a view to prove the feasibility of a self-configurable agent, the Branch and Bound mapping was implemented using Street language. The language is flexible enough to write agents in several approaches. Here, the problem is solved by dividing the agent into several entities and constructing production rules to define how these entities affect each other. This work also serves as a test agent for the purpose of measuring the performance of the mapping techniques discussed in Sections 5.4 and 5.5.

5.6.1 Entity relationship

The Branch and Bound mapping is defined using a number of entities, which are the data structures in working memory. Some of them are temporary ones used to determine the following major entities. These are not discussed here. These are also important to support the major entities. Each entity has several attributes to define its characteristics. Again, there are some temporary intermediate attributes, which are also not mentioned here. The major entities are as follows:

Global stores information that is shared among entities. It has the attributes to hold the number of PEs, routers, traffic edges in traffic graph, links in the network structure, minimum UBC, best mapping cost etc.

PE represents a processing element that realises a production rule. It contains the attributes of PE index, total incoming and outgoing traffics and its ranking based on total traffic.

Router contains the attributes of router index, position (row and column in 2D a mesh) and status, whether it is occupied or unoccupied.

Mapping relation indicates the relation between each PE and router. If there are N_P PEs and N_R routers, then the cardinality of the mapping relation is $N_P \times N_R$.

Mapping node is a node of the Branch and Bound search tree. It has the attributes of current cost, LBC and UBC. It keeps track of the PEs and routers not yet mapped. This entity uses some temporary attributes to calculate its major attributes. These temporary attributes are used to calculate total traffic between unmapped PEs, between mapped and unmapped PEs, the minimum distance between unmapped routers, the nearest router distance from mapped routers, and so on.

The relationships between these entities are shown in the entity relationship diagram in Figure 5.5. PE entities are related to each other and keep track of the total traffic between them, as derived from the traffic graph. Routers, in the same way, are related to other routers and store the distance between them obtained from the architecture graph. All PEs and routers are in relations through mapping relation entities. Each mapping node entity contains several mapping relationships to represent its partial mapping. It also keeps track of the unmapped PEs and unoccupied routers. The values of the global attributes are shared and updated during runtime. For example, if the UBC of a node is found to be less than the minimum UBC, the value of the global attribute representing the minimum UBC is updated accordingly.

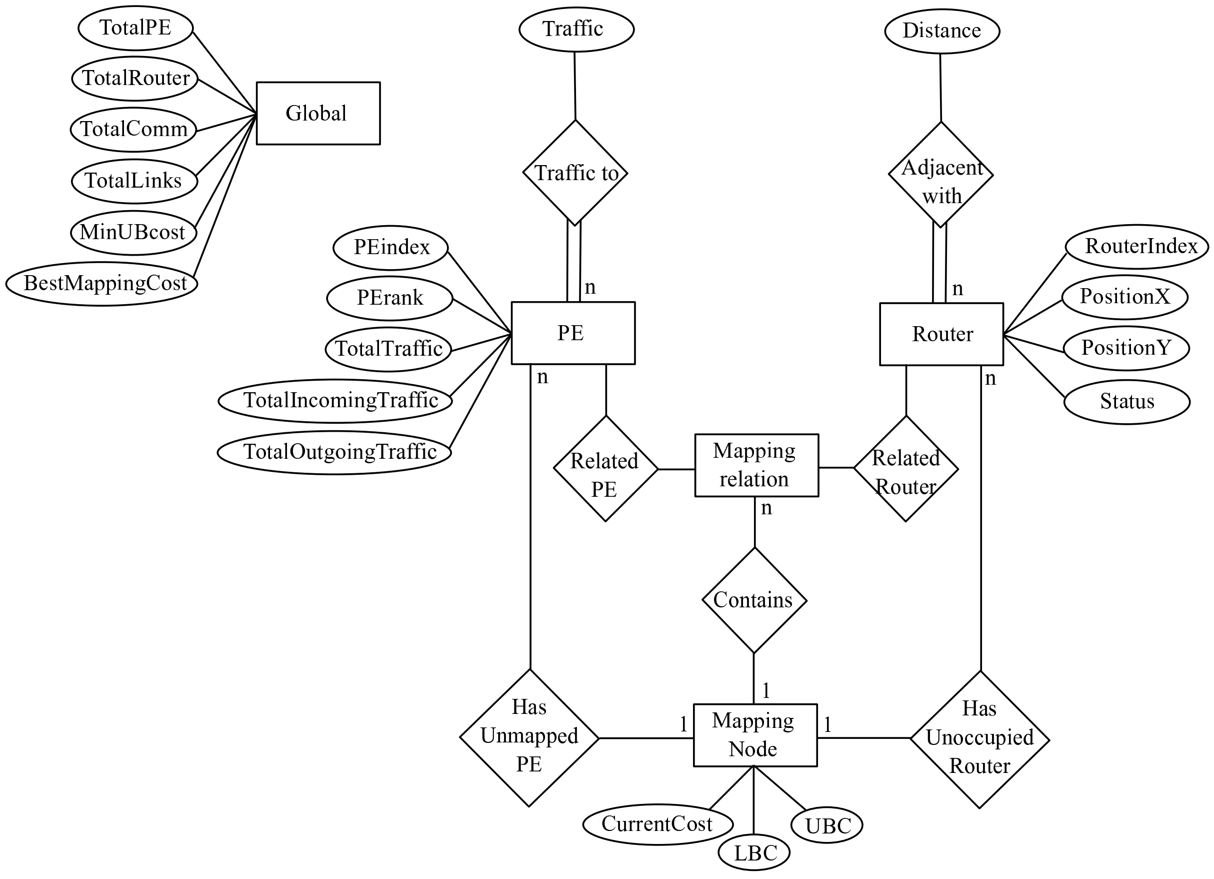


Figure 5.5. Relationship between the major entities. The rectangles, oval and diamond shapes indicate the entities, attributes and relationships respectively

5.6.2 Production rules

The agent to perform Branch and Bound mapping is written with 70 production rules in Street language. The definitions of the Street rules are available at the website of the Street project (Phillips 2016). Since Street language is very primitive by nature, it requires many intermediate rules to do simple calculations. For example, in order to check if variable `<var1>` is greater than variable `<var2>`, first one production rule calculates the difference of the variables, `<diff> = <var1> - <var2>`, and then another rule compares the variable containing the difference, `<diff>` with zero. This is because, Street language does not support direct comparison of variables. Table 5.1 lists some of the important Street rules used to realise the self-configurable agent.

Table 5.1. Important Street rules to realise the self-configurable agent.

Production rules	Description
calculateDistance	Calculates distance between routers
calculatePEtoPEtraffic	Calculates total traffic between PEs
calculateTotalLoad	Calculates total traffic by adding up incoming and outgoing traffics
getAllRelations	Generates all possible mapping relations between PEs and routers
createNewNodes	Creates child nodes from parent nodes if the node is expandable
getCurrentMappingCost	Calculates the partial mapping cost of mapping nodes
getLowerBoundCost	Calculates LBC of mapping nodes
generateGreedyMapping	Greedily maps unmapped PEs to unoccupied routers
getUpperBoundCost	Calculates UBC of mapping nodes
getMinUBCost	Calculates minimum UBC
getBestMappingCost	Finds the best mapping cost comparing current cost with it

5.7 Experiments

The self-configurable Street agent consists of 70 production rules. Each of these rules was associated with its own PE to execute it. The agent generated 10,387 dependencies between its PEs during execution. The traffic graph, shown in Figure 5.6, was derived from the dependencies. The PEs are represented by their indices and the weights of edges indicate the traffic between them. PE indices corresponding to relevant production rules are listed in Table A.1 of Appendix A.

Some experiments were conducted on this test agent to optimise the mapping of its PEs onto a 9×9 NoC structure using BB and SA techniques. First, the SA based mapping was applied to the traffic graph. The energy cost of the mapping configuration was calculated using (5.6) with the values of $E_s = 0.284$ picojoule, $E_l = 0.449$ picojoule and $E_b = 1.056$ picojoule obtained from bit energy model of (Hu and Marculescu 2003a). Figure 5.7 presents the energy cost at every iteration of SA. It can be observed that the mapping occasionally accepted bad moves to escape from local minima. Eventually, it levelled off at 1.71065×10^7 picojoule. Even more iterations did not provide any further decrease of the energy cost. The simulation took 1079 seconds on a Ubuntu virtual machine having 2 processor cores and 2 GB memory.

Figure 5.8 shows the 9×9 NoC structure with an indication of traffic associated with each router. Each small square block in this heatmap represents the traffic to and from

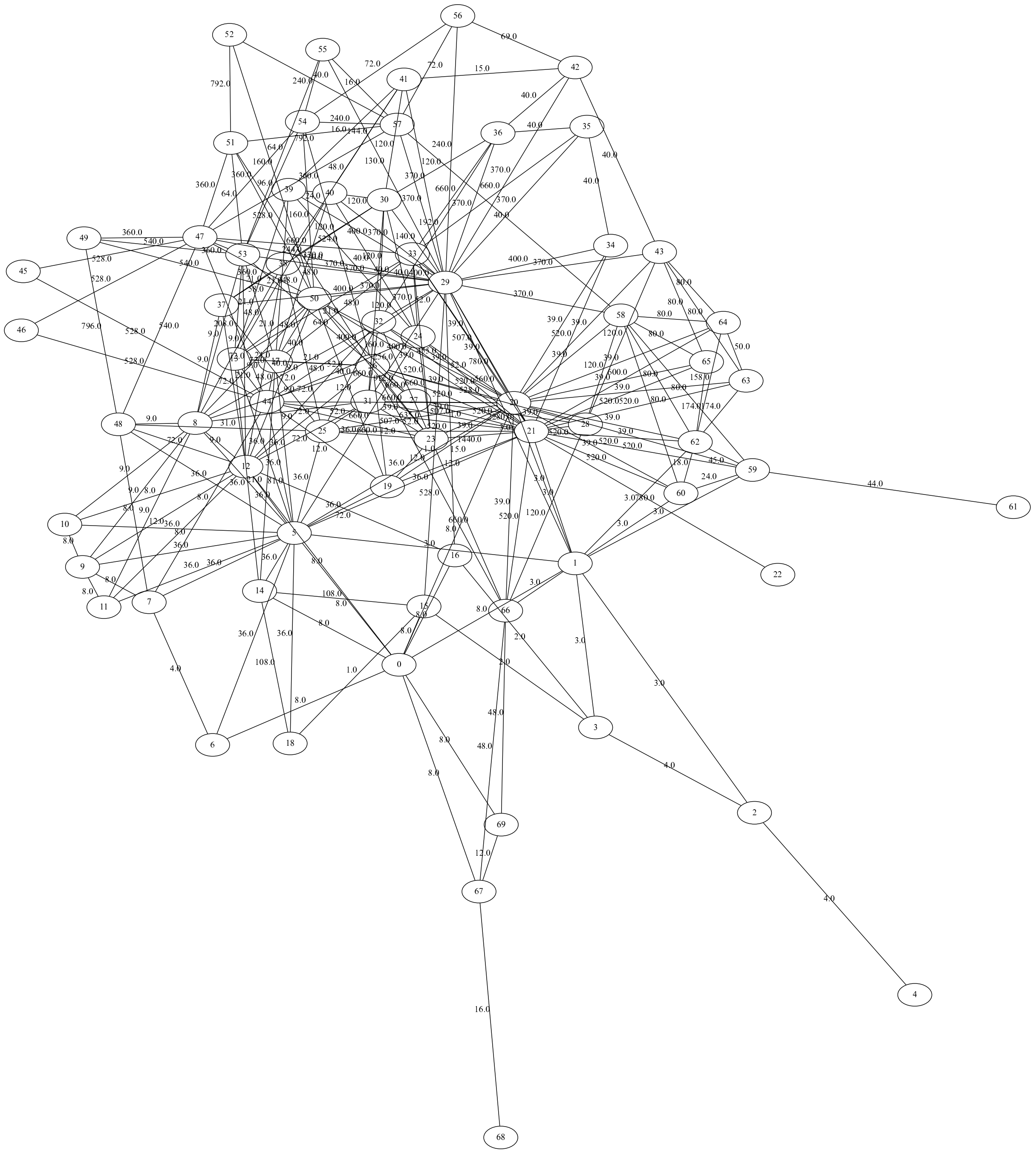


Figure 5.6. Traffic graph of the agent solving Branch and Bound mapping.

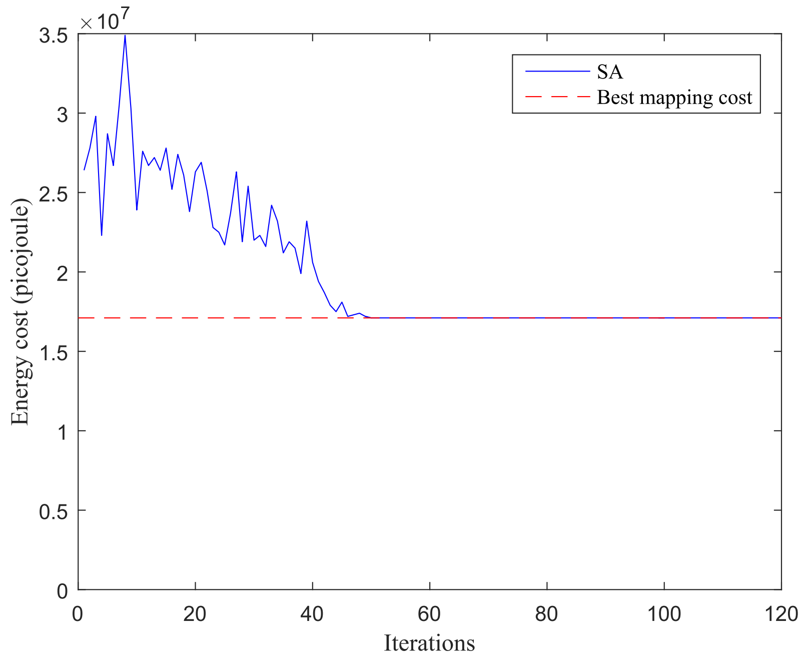


Figure 5.7. Energy cost in each iteration of Simulated Annealing.

the PE attached to each router. It can be observed from this figure that, as expected, the PEs with higher traffic concentration are mapped closer to each other.

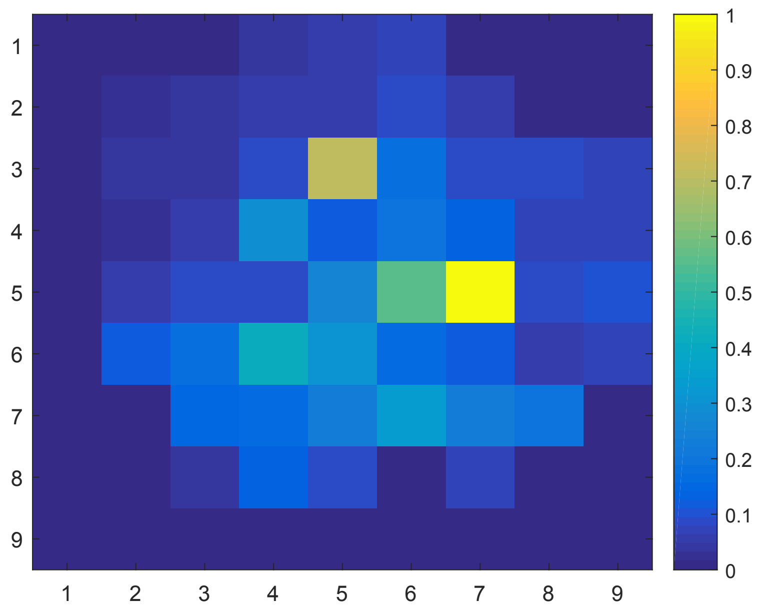


Figure 5.8. Router-wise traffic after Simulated Annealing mapping.

The BB mapping was also applied on the traffic graph. It generated a search tree having 5,439,129 nodes. For each node, the partial mapping cost, LBC and UBC were calculated. The minimum UBC and maximum LBC were tracked to limit the cost of the best mapping. Figure 5.9 illustrates the minimum UBC and maximum LBC limiting the best mapping cost. LBC and UBC of a node were analysed to check if the node was legal to be expanded based upon the principles mentioned in Section 5.5. If the node was not expandable, it was deleted. In this experiment, approximately 95% of the total nodes were deleted making BB an efficient optimisation method. The simulation took 203 seconds which was almost 5 times faster than SA, on the same system. The energy cost of the best mapping was measured 1.63669×10^7 picojoule. Figure 5.10 shows the traffic heatmap to and from the PE attached to each router after BB mapping.

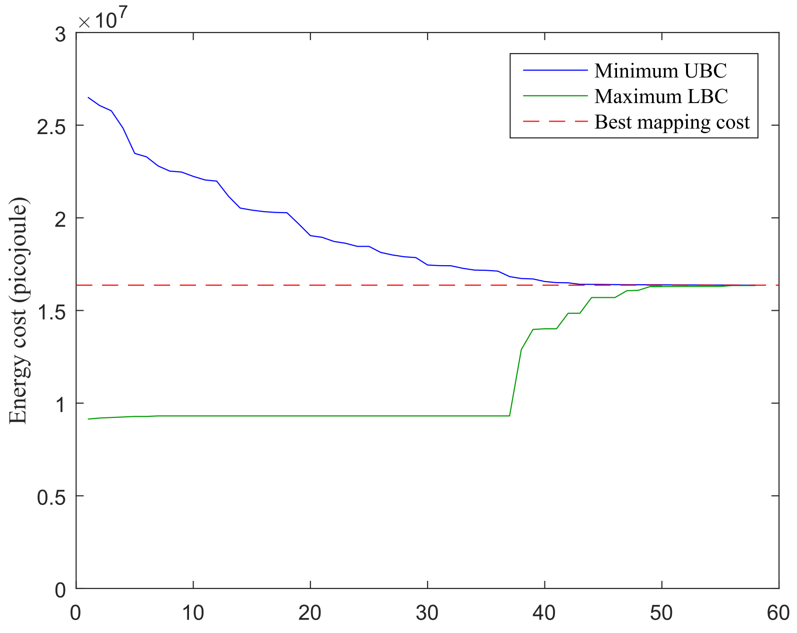


Figure 5.9. Minimum UBC and maximum LBC limiting the best energy cost.

Note that the difference between the energy costs of the mapping solutions obtained by SA and BB techniques is very negligible. To investigate how they scale in performance and efficiency with network sizes, both techniques were used to map synthesised network structures ranging from 4 to 196 routers. For every architecture, 10 random dependency sets were artificially generated. Each dependency set was used for mapping using both techniques. Figure 5.11 shows that the difference in energy costs of the optimum solutions for a particular network size for the mapping techniques was less than

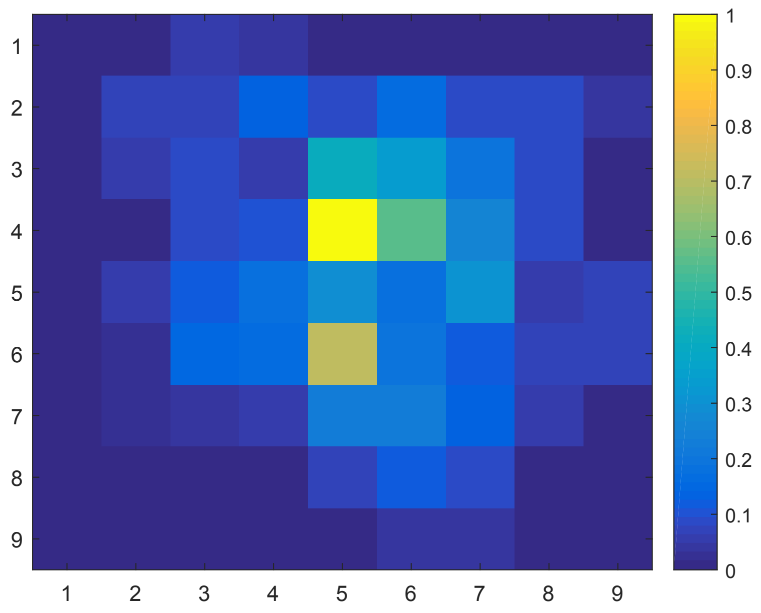


Figure 5.10. Router-wise traffic after Branch and Bound mapping.

8%. But the comparison of the computation time, as shown in Figure 5.12, demonstrates that BB outperformed SA by reaching the optimum solutions much faster. This is because SA blindly explored the solution space, whereas BB systematically deletes most of its unpromising solutions without exploring them in detail.

5.8 Analysis of Branch and Bound mapping

The BB algorithm finds a mapping solution through a very detailed search. Compared to SA, the advantage of BB mapping is its computation speed. However, when the number of PEs is very high, the nodes in its search tree have a high branching factor and its search space grows exponentially even when efficient bounding techniques are used. Although it trims the unpromising nodes of the search tree to speed up the process, this is not enough to obtain a mapping algorithm that is fast enough. It still needs to traverse the leaf nodes of the tree, which causes large memory consumption and still high processing time. It has to decide how much of the tree is to be deleted in order to save memory (Bader *et al.* 2005). Moreover, the implementation of the UBC and LBC calculation does not guarantee the best mapping configuration (Radu and VinŢan 2013). The UBC computation relies on a greedy mapping technique that quickly finds a mapping by trying to place unmapped PE on an unoccupied router

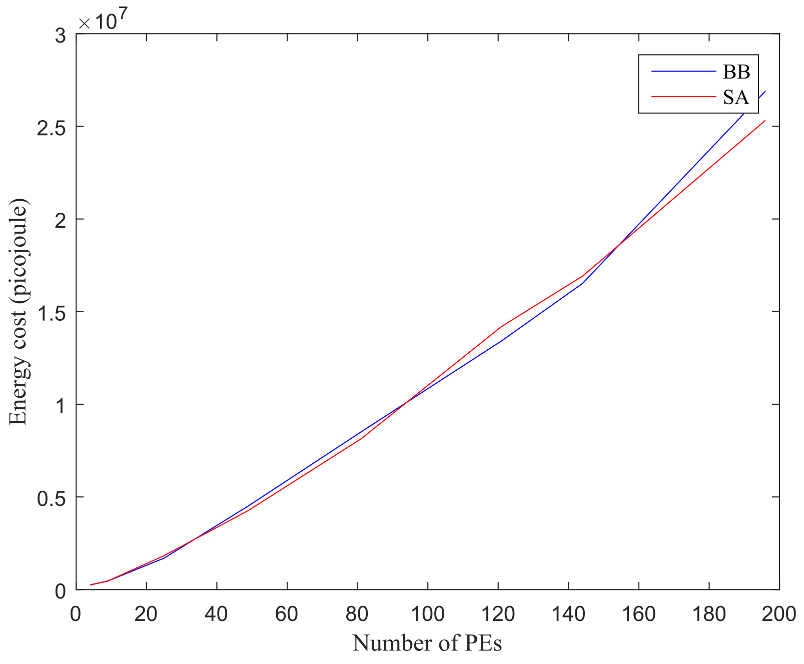


Figure 5.11. Energy cost comparison between SA and BB mapping.

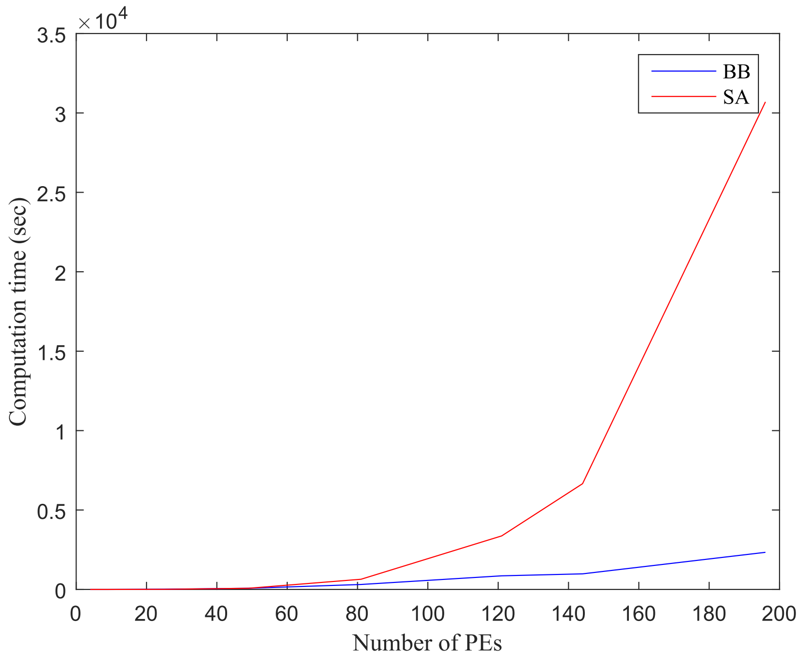


Figure 5.12. Computation time comparison between SA and BB mapping.

so that the energy cost is minimised. Making this kind of local optimal decisions does not necessarily lead to a global optimum. Although the greedy method provides some speed-up, this might not result in the best mapping. Furthermore, LBC is also not realistically computed. This maps an unmapped PE to the best unoccupied router without considering whether another PE has been already mapped to that router. Once again, the approach is fast but it does not provide the tightest LBC. The calculations of UBC and LBC can be made more realistic by checking whether a PE is already mapped to the best unoccupied router. If so, then the next best unoccupied router should be considered. The effect of this measure on overall performance of NoC demands further investigation.

5.9 Conclusion

This chapter has described two techniques for mapping homogeneous PEs to NoC routers based on Simulated Annealing, and Branch and Bound algorithms due to their superiorities over other candidate algorithms. A test case of a self-configurable agent was built to evaluate the performance of these mapping techniques. The agent mapped PEs onto a NoC structure based on BB algorithm using Street language. This establishes the capability of the Street Processor to implement a self-configurable agent.

When executed, the agent generated 10,387 dependencies between 70 PEs. These dependencies were analysed to map PEs onto a 9×9 mesh structure using both mapping techniques. The comparison shows that BB mapping reached a solution approximately 5 times faster than SA. The reason is that the searching process of SA was blind, and much time was wasted on exploring unpromising solutions, whereas BB mapping systematically deleted them by analysing the partial costs, LBC and UBC. In the test case of the self-configurable agent, it deleted almost 95% of its solution space without further exploring them. This result is supported by the experiments carried out on synthetic dependency sets where a BB approach provided significantly better performance in terms of computation time.

However, the computation time advantage of BB mapping comes at the cost of high memory requirement. Also, there is scope to make UBC and LBC calculations more realistic. Considering these issues, we consider SA to be more appropriate for applications where memory consideration is more important than computation time. The Street Processor, which usually contains a large number of PEs, is a suitable application of SA mapping. The processor also has the ability to periodically re-organise the mapping during sleep period for improved performance.

In the next chapter, efforts have been taken to reduce the computation time of SA by taking advantage of graph partitioning and a priority-based initial mapping.

Chapter 6

Priority-based Simulated Annealing of Grouped PEs

THIS chapter proposes a network on chip mapping technique for homogeneous systems. It uses graph partitioning to group the heavily-communicating processing elements and then map each group of PEs to a router. This reduces the amount of traffic that must pass through the network. In addition to this, the search space of the mapping problem is significantly reduced by this strategy. An improved Simulated Annealing-based mapping approach is proposed here, which prioritises the PEs and NoC routers based on their dependencies and positions respectively. These priorities are used to determine a heuristic initial mapping, which in turn results in a reduction in the computation time to find an optimum solution without sacrificing mapping quality.

6.1 Introduction

A large number of PEs makes it challenging to find an optimum PE-to-router mapping in a NoC because the search space grows exponentially with the number of PEs. For homogeneous systems such as the Street Processor, where the number of PEs may range up to thousands, a heuristic mapping technique must be used to solve the mapping problem.

The search space can be significantly reduced using graph partitioning algorithms. The classical graph partitioning problem is to divide the vertices of a weighted graph into approximately equal partitions such that the number and weights of edges connecting vertices assigned to different partitions is minimised. This finds applications in many areas including parallel computing, task scheduling and VLSI design (Cheng and Wei 1991, Gilbert and Zmijewski 1987, Garbers *et al.* 1990). It is a NP-complete optimisation problem that may have prohibitive computation time requirements. There exists a critical trade-off between time and performance (that is, the quality of the partitioning obtained) of the partitioning algorithms. Many algorithms have been developed to find reasonably good partitions (Kernighan and Lin 1970, Fiduccia and Mattheyses 1982, Hendrickson and Leland 1995, Hagen and Kahng 1991).

NoC mapping techniques based on application partitioning have been proposed by (Tosun 2011, Jang and Pan 2010). Tosun (2011) observed that integer linear programming (ILP) provided optimum mapping; but since it searched for every possible solution in the huge solution space, it took a very long time to determine the optimum solution. He proposed a clustering-based relaxation technique for ILP formulations in order to reach the optimum result within tolerable time limits. Jang and Pan (2010) proposed a partition-based mapping for an irregular mesh network. However, both of these techniques did not allow inter-partition movement of PEs. This caused optimised mapping inside partitions but sacrificed global optimisation. Sahu *et al.* (Sahu *et al.* 2010) proposed a mapping algorithm based on Kernighan-Lin (KL) partitioning (Kernighan and Lin 1970), which is a bi-partitioning algorithm. This approach considered swapping of PEs between partitions in an iterative way.

An improved Simulated Annealing algorithm was proposed in (Radu and VinŢan 2013), which optimised mapping by clustering PEs implicitly during the swapping process. However, the authors in (Radu *et al.* 2013) concluded that an evolutionary algorithm performs better than this. Particle swarm optimisation was proposed by Sahu *et al.* (2014), which used a deterministic initial mapping to explore the search space. This initial mapping technique was unable to obtain a good mapping as NoC size increased. A cluster-based Simulated Annealing was proposed in (Lu *et al.* 2008) to provide runtime improvements without compromising the quality of the solution, compared to the original Simulated Annealing. However the technique proposed in (Radu and VinŢan 2013) proved to produce a better quality mapping even without generating any initial mapping.

A cluster-based NoC architecture was also described in (Modarressi and Sarbazi-Azad 2012) but this introduced additional configuration switches between the routers, which caused a 10% to 35% area overhead. Moreover, generally the routers in a NoC are expensive. In Intel's 80-core TeraFlops, for example, more than 80% of the on-chip communication power was consumed by routers (Oracle 2012). Hence, it is desirable to reduce the number of routers in a NoC structure containing a large number of PEs. However all of the above mentioned mapping techniques used partitioning (or clustering) to localise PEs only, and they mapped one PE to each router. As a result, they did not aid in reducing the number of routers.

In this chapter, we describe a method for partitioning PEs into groups of PEs (GPEs) using a multi-level partitioning algorithm. The number of required routers is reduced by assigning a GPE to a router, instead of a single PE. The PEs within a GPE communicate with each other using the crossbar of the router, as discussed in Section 4.4. The mapping of GPEs to routers is done by Simulated Annealing. In the second phase of this work, we propose a further improvement to the performance of Simulated Annealing by generating a good initial mapping in order to converge faster to the optimum solution. This is done by prioritising the routers and GPEs based on their locations and inter-dependencies respectively. Our experiments show that all these efforts significantly reduce the computation time to find the solution without degrading the performance of the mapped network of PEs.

6.2 Graph partitioning

Graph partitioning is a method of dividing the vertices of a weighted graph into roughly equal partitions or groups. The number of edges whose incident vertices belong to different partitions is called the edge-cut of the partition. A good partitioning technique minimises the edge-cuts, and the weights of the edges connecting vertices in different partitions.

A weighted graph can be defined as $G = (V, E, W_E)$ where V, E and W_E respectively denote the sets of vertices, edges, and the function to map the edges to their weights, i.e. $W_E : E \to C$, where C is the set of all possible edge weights. The k-way partitioning of the graph is defined as to partition V into k subsets such that $\bigcup_{i=1}^{k} V_i = V$ and $V_i \cap V_j = \emptyset, i \neq j$ and the sum of edge weights whose whose incident vertices belong to different subsets of V is minimised. Figure 6.1 illustrates an example of a graph partitioning where the graph in (a) is partitioned into the graph in (b).

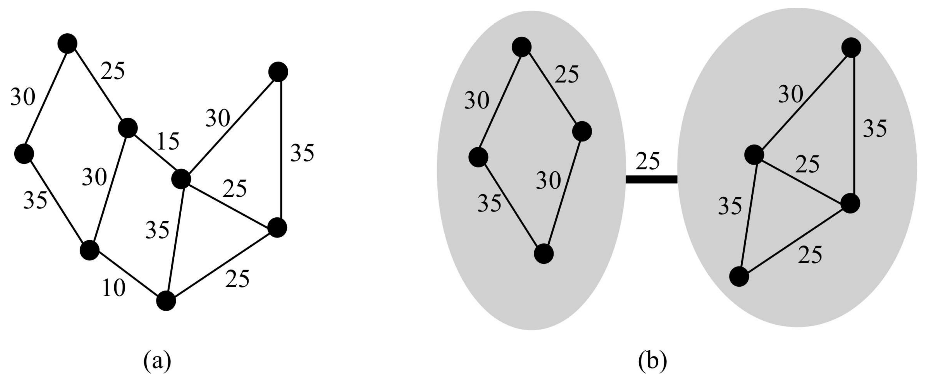


Figure 6.1. Example of graph partitioning. The weights are shown for corresponding edges

In this work, graph partitioning is applied on the traffic graph (introduced in Section 5.3.2) to divide it into a number of GPEs. In addition to evenly distributing traffic over the network, it provides some additional benefits as well. First, it minimises the search space of the mapping problem, which helps to reduce the computation time. Secondly, it reduces the number of required routers, as instead of a single PE, each router is associated with one GPE comprising multiple PEs.

6.2.1 Multi-level graph partitioning

Graph partitioning is a NP-complete optimisation problem. Various approaches have been taken to find a good partitioning in reasonable computation time. Spectral partitioning methods (Pothen *et al.* 1990) produce excellent partitions for a wide class of problems. However, these methods are very time consuming. The multi-level spectral bisection algorithm (Barnard and Simon 1993) speeds up the spectral partitioning methods by an order of magnitude without any loss of quality of the edge-cut. But this method also takes a large amount of time. Geometric partitioning algorithms (Miller *et al.* 1991, Heath and Raghavan 1995) tend to be fast but often yield partitions that are worse than those obtained by spectral methods. Compared to spectral and geometric partitionings, multi-level graph partitioning (Hendrickson and Leland 1995, Karypis and Kumar 1995) provides good quality partitions in a short period of time. This work uses this method to partition the traffic graph.

Multi-level partitioning is a straightforward approach to approximate the graph partitioning problem. It consists of three phases: graph coarsening, initial partitioning, and uncoarsening. During the coarsening phase the graph is gradually reduced to coarser graphs using local operations. In the initial partitioning phase the coarsest and therefore smallest graph can be partitioned using a suitable partitioning algorithm. This partition is then applied in the refinement phase to each larger graph and further refined. The phases are illustrated in Figure 6.2.

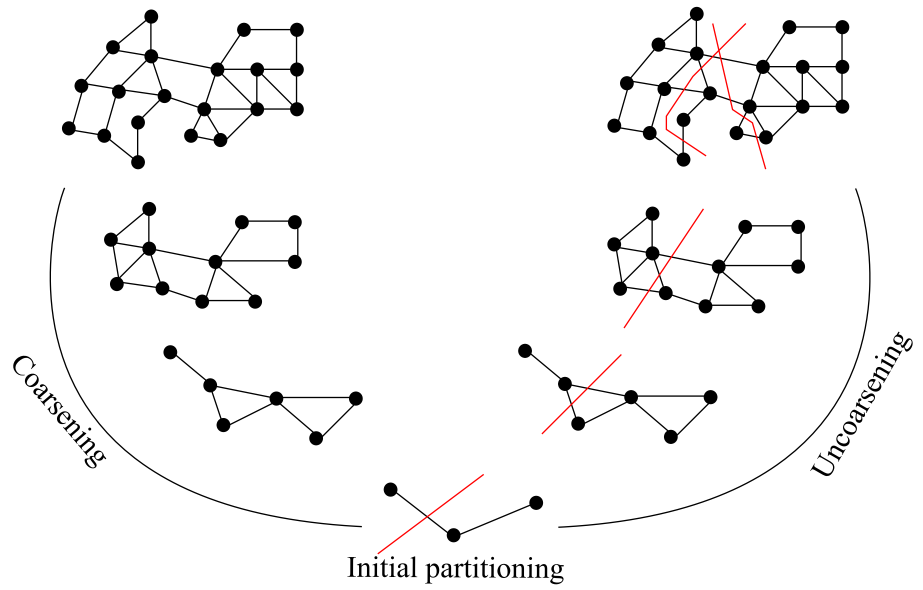


Figure 6.2. The phases of multi-level graph partitioning.

Coarsening: In the graph coarsening phase, a series of successively smaller graphs is derived from the input graph. The fundamental step of coarsening is the edge collapse operation. In this step, two vertices joined by an edge are merged, and the new vertex retains edges connecting to the union of the neighbours of the merged vertices. Mathematically, collapsing an edge $e = (v_1, v_2) \in E$ into a vertex $v_3 \notin V$ results in a graph $G' = (V', E', W'_E)$, where $V' = V \setminus \{v_1, v_2\} \cup \{v_3\}$. For E', at first, the edge $e = (v_1, v_2)$ is removed from E, and then the set of edges $e' = \{(v_3, v_k) | v_k \in N(v_1) \cup N(v_2)\}$ is added. Here $N(v_i) = \{v_k | (v_i, v_k) \in E\}$ represents the sets of neighbours of v_i. The weights of edges are left unchanged unless both merged vertices are adjacent to the same neighbour. In this case, the new edge that represents the two original edges is given a weight equal to the sum of the weights of the two edges it replaces. (6.1) gives the equation to calculate edge weights. Figure 6.3 demonstrates two examples of edge collapse. This edge collapsing process continues until the coarsened graph contains a small number of vertices, yet enough to perform a meaningful initial partitioning. In case of partitioning the traffic graph of the Street Processor, considered later in Section 6.2.2, the process continues until the size of the graph is reduced to 100 vertices.

$$W_E((v_3, v_k)) = \begin{cases} W_E((v_1, v_k)) + W_E((v_2, v_k)) & \text{if } (v_3, v_k) \in e' \text{ and } v_k \in N(v_1) \cap N(v_2) \\ W_E((v_l, v_k)) & \text{if } (v_3, v_k) \in e' \text{ and } v_k \in N(v_l), v_l \in \{v_1, v_2\} \\ W_E((v_3, v_k)) & \text{else} \end{cases} \tag{6.1}$$

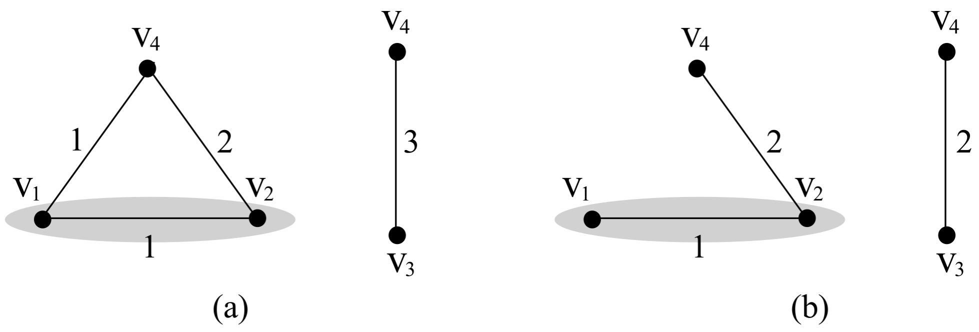


Figure 6.3. The collapse of an edge. Vertices v_1 and v_2 are collapsed into the new vertex v_3. Note that the edge weights of the edge $\{v_3, v_4\}$ in (a) and (b) are measured by first two cases of (6.1) respectively

Initial partitioning: In this phase, a partitioning of the coarsest graph is computed using the KL algorithm. Since the coarsest graph is usually very small, this step is very fast. This is done by by recursively calculating k-way bi-partitions of the graph (Simon 1991). To calculate a bi-partition, first two pseudo peripheral vertices are found, i.e., two vertices having the greatest distance to each other. This can be done by starting a breadth first search at one random vertex v_0. Then a vertex v_1 is selected with maximal distance to v_0. Now a second breadth first search is started at v_1, and a vertex v_2 with maximal distance to v_1 is selected. This process is repeated until the distance between v_i and v_{i+1} stops increasing.

Each vertex is then assigned to that pseudo peripheral vertex it is closer to. This algorithm is used in many graph partitioning tools such as METIS (Karypis and Kumar 1998).

Uncoarsening: In the uncoarsening phase, the partitioning of the smallest graph is projected to the successively larger graphs by assigning the pairs of vertices that were collapsed together to the same partition as that of their corresponding collapsed vertex. After each projection step, the partitioning is refined using boundary KL refinement algorithm (Karypis and Kumar 1998) to iteratively swap vertices between partitions as long as such moves reduce the edge-cuts. The refinement process considers the boundary vertices of the partitions only. The internal vertices may become boundary vertices if their adjacent boundary vertices are swapped between partitions. This uncoarsening phase ends when the partitioning solution has been projected all the way to the original traffic graph.

6.2.2 Multi-level partitioning of traffic graph

This work uses the tool METIS (Karypis and Kumar 1998) for a quick multi-level partitioning of traffic graph. The quality of the partitionings, i.e. the weights of edge-cuts between partitions, produced by this tool are on the average 6%-23% better than those produced by other state of the art schemes, according to the authors. METIS uses novel approaches to successively reduce the size of the graph in the coarsening phase. During this time, METIS employs algorithms that make it easier to find a high-quality partition of the coarsest graph. The use of powerful coarsening schemes also allows the refinement process to be simplified considerably, making the multi-level scheme quite fast. During refinement, it focuses primarily on the portion of the graph that is close to the partition boundary.

These highly tuned algorithms allow METIS to produce high-quality consistent partitions in a small amount of time.

METIS is applied on the traffic graph $TG = (\mathbb{P}, \mathbb{T}, W_{\mathbb{T}})$, which has been defined in Section 5.3.2. Here, $\mathbb{P}$, $\mathbb{T}$ and $W_{\mathbb{T}}$ respectively represent the sets of PEs, communication edges and the function to map the communication edges to the total traffic between them. It partitions the $\mathbb{P}$ into a set of k GPEs, $\mathbb{G} = \{G_1, G_2, ...G_k\}$, where $\bigcup_{i=1}^{k} G_i = \mathbb{P}$. Then total traffic between two GPEs G_i and G_j ($i \neq j$) can be calculated by (6.2)

$$W_{\mathbb{Q}}((G_i, G_j)) = \sum_{p_i \in G_i, p_j \in G_j} W_{\mathbb{T}}((p_i, p_j)) \tag{6.2}$$

Using this equation, a coarser traffic graph that defines the traffics between GPEs is derived. This traffic graph, referred as partitioned traffic graph (PTG), can be defined as $PTG = (\mathbb{G}, \mathbb{Q}, W_{\mathbb{Q}})$, where $\mathbb{G}$, $\mathbb{Q}$ and $W_{\mathbb{Q}}$ represent the sets of GPEs, edges between GPEs and the function from the set of edges to the set of possible weights respectively.

6.3 Priority-based Simulated Annealing

Each GPE is considered to be mapped to a NoC router in this proposed mapping technique. The Simulated Annealing algorithm, described in Section 5.4, blindly explores many ultimately unusable paths, which consumes time unnecessarily. We improve this by heuristically generating an initial mapping considering the *priorities* of GPEs and routers. The priorities are derived by analysing the partitioned traffic graph and architecture graph respectively. This algorithm also involves a localised swapping technique, in contrast to a uniform random swap used in general Simulated Annealing. These efforts make this Priority-based Simulated Annealing (PSA) technique converge faster to solution. Figure 6.4 shows the pseudo-code of the proposed mapping technique. The major steps of PSA are described in the following subsections.

6.3.1 Initial mapping

The Priority-based Simulated Annealing mapping technique generates an initial mapping using the PTG and AG. The initial mapping consists of three steps: GPE prioritising, router grouping and GPE-to-router mapping.

GPE prioritising: The GPEs are prioritised based on their total number of connections (the number of attached edges) and total traffic demand (the sum of the weights of the attached edges). The priority of a GPE $G_k \in \mathbb{G}$, depends on the number of attached edges n_k and total traffic C_k associated to it. The total traffic of G_k is calculated as

$$C_k = \sum_{i=1, i \neq k}^{n_k} W_{\mathbb{Q}}((G_i, G_k)) \tag{6.3}$$

```
Require: $PTG$, $AG$, $T_0$, $N_R$
1: Sort GPEs $\{G_1, G_2, ...G_k\}$ of $PTG$ by traffic demand
2: $T = T_0$
3: $M = InitialMapping(PTG, AG)$
4: $C = M.GetMappingCost()$
5: $S_{max} = AG.GetMaxStage()$
6: for $s = 1$ to $S_{max}$ do
7:     $done = false$
8:     while $done \neq true$ do
9:         for $i = 1$ to $N_R$ do
10:            $M_{new} = LocalisedSwap(M, s)$
11:            $C_{new} = M_{new}.GetMappingCost()$
12:            $\Delta C = (C_{new} - C)/C$
13:            if $\Delta C < 0$ or $F(\Delta C) \geq random(0,1)$ then
14:                $M = M_{new}$
15:                $C = C_{new}$
16:            else
17:                $M = RollbackSwap(M_{new})$
18:            end if
19:        end for
20:        if $StopTest(C, C_{prev})$ then
21:            $M_{best} = M$
22:            $C_{best} = C$
23:            $done = true$
24:        else
25:            $T = T \times 0.9^{\lfloor i/N_R \rfloor}$
26:            $C_{prev} = C$
27:        end if
28:    end while
29: end for
30: return $M_{best}$
```

Figure 6.4. Priority-based Simulated Annealing algorithm for GPE mapping.

where $W_{\mathbb{Q}}(G_i, G_k)$ is total traffic between G_i and G_k over a period of time. Generally, the GPE with the higher number of edges has the higher priority. If two GPEs have same number of edges, then the total traffic is compared. The GPE with higher traffic demand gets higher priority in this case.

Let us consider two GPEs G_1 and G_2 are connected to n_1 and n_2 number of other GPEs, and the total traffic associated to them are C_1 and C_2 respectively. If $n_1 > n_2$, or $(n_1 = n_2$ and $C_1 > C_2)$, then G_1 is considered to have higher priority than G_2. This is because, the GPEs with more edges and higher traffic demands affect the overall traffic of the network more compared to other GPEs.

Router grouping: The routers are grouped based on their degrees of adjacency and global distance, which are measured by their positions and distances from other routers. If a router r_k has a_k number of adjacent routers, then the degree of adjacency of the router is a_k. The value is important because a router with a higher degree of adjacency has a larger communication capability, so it should be assigned to GPEs with higher connectivities. On the other hand, the degree of global distance reflects the global connectivity, and is calculated by the sum of distances from this router to other routers. If the sum of Manhattan distances from r_k at position (x_k, y_k) to all other routers in the network is g_k, then its degree of global distance is considered g_k. It is calculated as

$$g_k = \sum_{i=1, i \neq k}^{N_R} (|x_i - x_k| + |y_i - y_k|) \tag{6.4}$$

where N_R is the number of routers and (x_i, y_i) is the position of other routers.

Two routers r_1 and r_2 having the degrees of adjacency a_1 and a_2, and degrees of global distance g_1 and g_2 respectively, are the members of the same group if ($a_1 = a_2$ and $g_1 = g_2$). If $a_1 > a_2$ or ($a_1 = a_2$ and $g_1 > g_2$), then r_1 is associated to a group having higher priority than that of r_2.

Figure 6.5 shows two examples of 5×5 and 6×6 network structures where each cell represents a router and its priority. It indicates that the center routers have the highest priorities, and the priorities gradually decreases as the position of the routers move away from the center.

GPE-to-router mapping: The GPEs are then mapped to routers according to their priorities. The high-priority GPEs are mapped to router groups with higher priorities. This satisfies the traffic demands of the high-priority GPEs as those routers

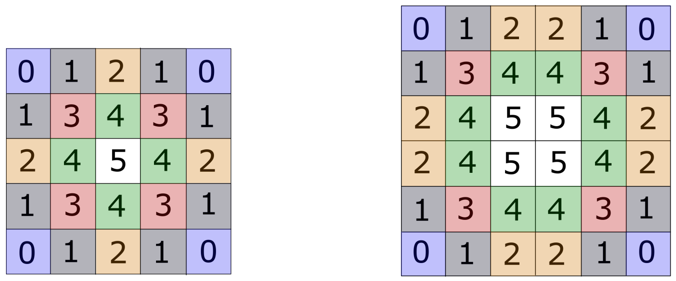


Figure 6.5. Routers indicating their priorities in 5×5 and 6×6 network structures.

provide more adjacency and they are relatively closer to other routers. If there is any remaining router in the current group, the GPE with the next priority is mapped into that. Otherwise, the GPE is mapped to the next router group, and so on. As a result, the GPEs will be initially concentrated nearer the centre, and will be gradually moved towards the edges.

6.3.2 Annealing

The PSA follows the principles of original Simulated Annealing with some modification. The GPEs mapped in a router group are allowed to move inside and outside of the group depending on the swapping distance during the localised swap. This makes the algorithm robust enough not to limit annealing inside a router group only. The annealing stages and localised swap are discussed below:

Annealing stages: The annealing continues over several stages. The stages dictates the swapping distances between the routers. Annealing at stage s corresponding to swapping distance d, means that the GPEs within distance $d = D - s + 1$ are allowed to be swapped in that stage. Here D represents the highest Manhattan distance between routers, which is essentially the diameter for a mesh structure. For a NoC structure of $n \times n$ routers, $D = 2 \times n - 2$. Annealing starts with the highest swapping distance D at stage $s = 1$, allowing the GPEs with $d \leq D$ to be exchanged. Then, it moves to the lower subsequent swapping distances. For example, in the second stage, $s = 2$, the GPEs with $d \leq (D - 1)$ may be swapped, and so on. In the final stage, on GPEs with swapping distance $d = 1$, i.e. only the neighbouring GPEs may be swapped.

```
Require: $M, s$
 1: $d = M.GetAllowableDistance(s)$
 2: $r_i = M.RandomRouterSelection()$
 3: $rg = M.GetRouterGroup(r_i)$
 4: $d_{max} = rg.GetMaxDistanceInGroup()$
 5: if $d \leq d_{max}$ then
 6:     $r_j = rg.IngroupRouterSelection()$
 7: else
 8:     $r_j = rg.RandomRouterSelection()$
 9: end if
10: $M_{new} = Swap(r_i, r_j)$
11: return $M_{new}$
```

Figure 6.6. Pseudocode of localised swap.

Localised swap: In the annealing stages, instead of a uniform random swap, PSA conducts a localised swap, which selectively allows GPE swapping inside and outside a router group. At each stage, a pair of routers is randomly selected within and/or across the group based on the value of d which is the allowable swapping distance at stage s. The first router r_i is selected randomly. Then the maximum swapping distance from r_i to its in-group routers, d_{max}, is calculated. If $d \leq d_{max}$, then the second router r_j is randomly selected from the same group; otherwise r_j is selected irrespective of router group boundaries, i.e. from inside or outside of the group r_i belongs to. Figure 6.6 shows the pseudocode of localised swap.

This process results in a blind swapping in the earlier stages, ending up with exclusively inside-group swaps in the later stages. This helps to escape from local minima during the earlier stages and limits the search to be carried out at near-optimal solutions in the later stages.

6.4 Test case: the Subsumption Cockroach agent

To compare the performance of the mapping techniques we applied them to an AI Cockroach agent developed by James Donovan, an undergraduate student at the University of Adelaide. The agent is reported in detail in the Street website (Phillips 2016). The structures of PEs and routers are discussed in Chapters 3 and 4, respectively. The agent uses a subsumption architecture (Brooks 1986, Brooks and Connell 1986). It is built using a number of modules that are realised in Street language. Considering that each rule is allocated to a single PE, the dependency and traffic graphs of the agent are generated. These graphs are used to map the PEs onto the network structure. Although a detailed discussion of the subsumption architecture is not within the scope of this thesis, it is necessary to discuss the basics of this architecture to understand the dependencies expressed in the Street production rules.

6.4.1 Subsumption architecture

A subsumption architecture stacks simple artificial intelligence behaviours on top of each other to create complex behaviour. The idea is that low level behaviours will continue to work without re-implementing them into each advanced level of behaviour. Subsumption architecture is an alternative to traditional AI, which usually consists of taking sensor inputs, analysing the data to produce a representation of the environment, and then deciding what output signals to send to actuators to respond to the environment. A subsumption architecture instead closely relates sensor readings to actions performed, with no complete analysis of the environment to decide what to do next. Instead, many layers of control have their own behaviour that incrementally affect the total behaviour of the system.

Levels of competence

This architecture focuses on a hierarchy of behaviours. The behaviour that has no reliances on other behaviours is a level 0 behaviour. A behaviour that relies only on the level 0 behaviour is level 1, and so on, until every desirable behaviour has a level assigned. Each level of desirable behaviour is termed a level of competence.

Layers of control

Each level of competence has a corresponding layer of control, which specifies modules and data-paths between them to satisfy the level of competence. The data-paths in a layer of control specify which module outputs connect to which module inputs. A layer of control cannot change the data-paths created by lower layers of control, but they can use inhibition and suppression mechanisms to alter the inputs and outputs of a module to achieve desirable behaviour. An inhibitor can be connected from the output of one module to the output data-path of another module. A suppressor is similar to an inhibitor, but is placed from the output of one module to the input data-path of another. Inhibitors and suppressors are used when a pre-established data-path from a lower layer of control must be altered to achieve a higher level of competence.

Modules

Modules are the functional parts of the architecture that consist of inputs, outputs, data storage and states. A finite state machine (FSM) moves through the states of outputs asynchronously to every other state. Only the latest signal is stored at the inputs, so if it is not analysed before the next signal arrives, it will be lost forever.

6.4.2 Development of the Cockroach agent

This agent demonstrates some aspects of the behaviour of cockroaches. It wanders around a room avoiding bright light, looks for food when it is hungry and goes to dark places to rest when it is tired. It retreats to the last dark place as preferred place, and likes to follow walls rather than open spaces. This subsumption Cockroach agent was built to be used as a test case to evaluate the performance of Street project. The correctness of the agent was verified by a simulation developed in the Java platform.

The following layers of competences specify the goal of each increment of functionality for the Cockroach agent.

Layer 0 corresponds to the first level of competence, in which the agent rotates to align itself approximately parallel to the wall. In this layer, it has three modules to detects walls, to determine the angle to align with a wall and to turn the agent.

Layer 1 adds the functionality that allows the Cockroach agent to move forward when not rotating. In addition to the modules of layer 0, it uses a module to move forward. The module moves the agent forward in small increments when it is not turning.

Layer 2 causes the agent to randomly depart from walls when following them. The module in this layer outputs a random angle at a random time when the agent is lined up with a wall.

Layer 3 provides functionalities of the agent to navigate to a dark place when tired and frightened, and sleep when tired. It has modules to check the levels of tiredness and fright, and to track the last dark place.

Layer 4 is responsible for providing the agent with a hunger level and the ability to seek and eat food previously found when hungry using its two modules.

6.5 Experiments

The modules of the Cockroach agent were implemented using Street language with 293 production rules, each realised on a single PE. When the agent was executed for approximately 20 minutes, it generated a set of dependencies between PEs. The traffic graph generated from these dependencies using the process described in Section 5.3.2, is shown in Figure 6.7, in which the PEs are indicated by their indices. The list of PE indices and corresponding production rules are listed in Table B.1 of Appendix B.

In order to compare the performance of the mapping techniques, the PEs of the Cockroach agent were mapped using the PSA, SA and BB mapping techniques. All of the experiments were carried out on an Ubuntu virtual machine having 2 processor cores and 2 GB memory. Since the mapping techniques are heuristic, all of the approaches did not necessarily produce the same solution. Figure 6.8 shows the computation time required for each of the techniques to reach their final solutions. The figure also includes the computation times for the mapping techniques in the case of the Street implementation of the self-configurable agent (discussed in Section 5.6). This is magnified in an inset of the figure for greater clarity. In both cases, PSA reached its solutions significantly faster than SA. For the self-configurable agent, PSA took approximately half of the computation time required by SA. For the Cockroach agent, the advantage of PSA was greater. In this test case, PSA worked 2.5 times faster than SA. It should also be noticed that the self-configurable agent had 70 PEs, so all of the mapping techniques took much less time, compared to the case of the Cockroach agent which used 293 PEs. But the computation time of PSA increased relatively more slowly than both SA and BB techniques. It increased 36 times from 564 seconds (for the self-configurable agent) to 20388 seconds (for the Cockroach agent) for PSA, whereas 50 times (1080 and 53497 seconds) and 57 times (203 and 11672 seconds) respectively for SA and BB.

Although PSA demonstrated a significant computation time improvement over SA, it still took a considerable amount of time, because the search space was still very

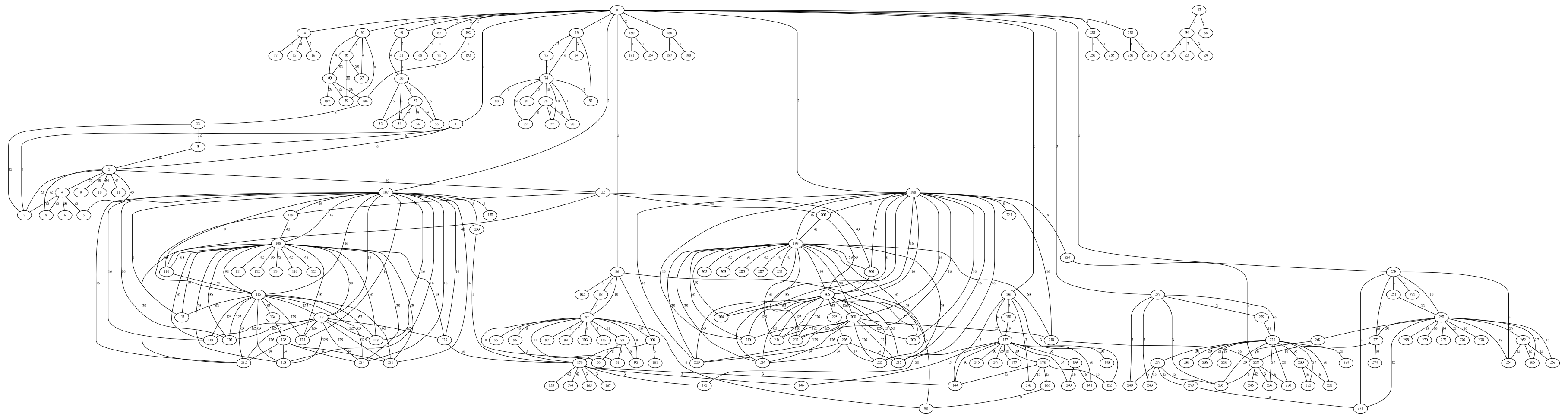


Figure 6.7. Traffic graph of the Cockroach agent.

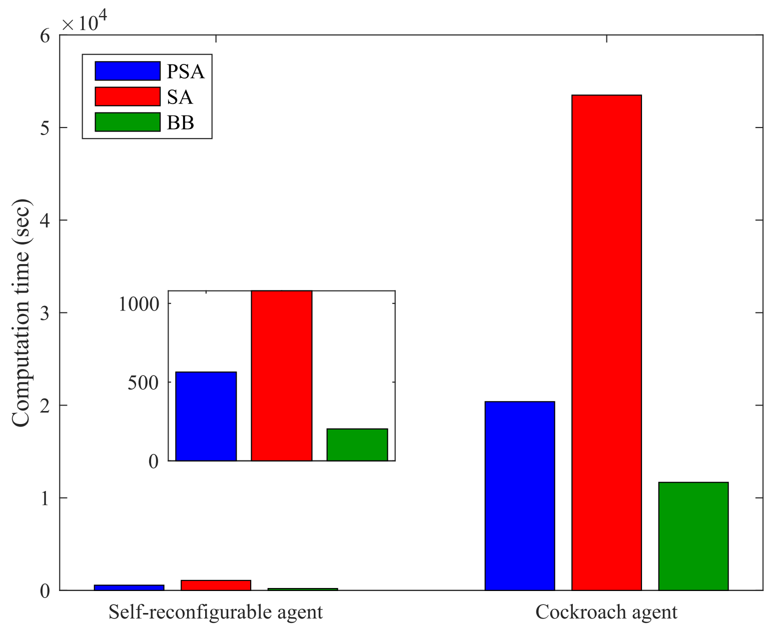


Figure 6.8. Computation time comparison between mappings of two Street agents. The self-configurable agent and the Cockroach agent contain 70 and 293 PEs respectively.

large. The traffic graph of Figure 6.7 exhibits some identifiable clusters of PEs. This suggests that a hierarchical grouping of PEs was a sensible approach. We applied the multi-level partitioning algorithm on the traffic graph to generate a partitioned traffic graph between the groups of PEs. Figure 6.9 shows the partitioned traffic graph of the Cockroach agent. It generated 27 GPEs optimising total edge weights between them. Here the GPE size and number were not imposed, rather a balanced distribution of the network traffic was considered, i.e. each GPE supported approximately same amount of internal traffic. The GPEs were then mapped to a 6×6 NoC structure using PSA and SA techniques.

Both of the techniques of SA and PSA were iterative processes. Figure 6.10 shows the energy consumption of the GPE-to-router mapping that was generated at each iteration, estimated using the energy model described in Section 5.3.1. Since PSA started annealing from a heuristic initial mapping, rather than random mapping, it reached the solution earlier than SA. The figure shows that it reached its solution at iteration 32, whereas SA reached its solution at iteration 71. To reach the solution, PSA took 36 seconds for computation and SA took 75 seconds. This clearly indicates that PSA converged to a solution much faster than SA.

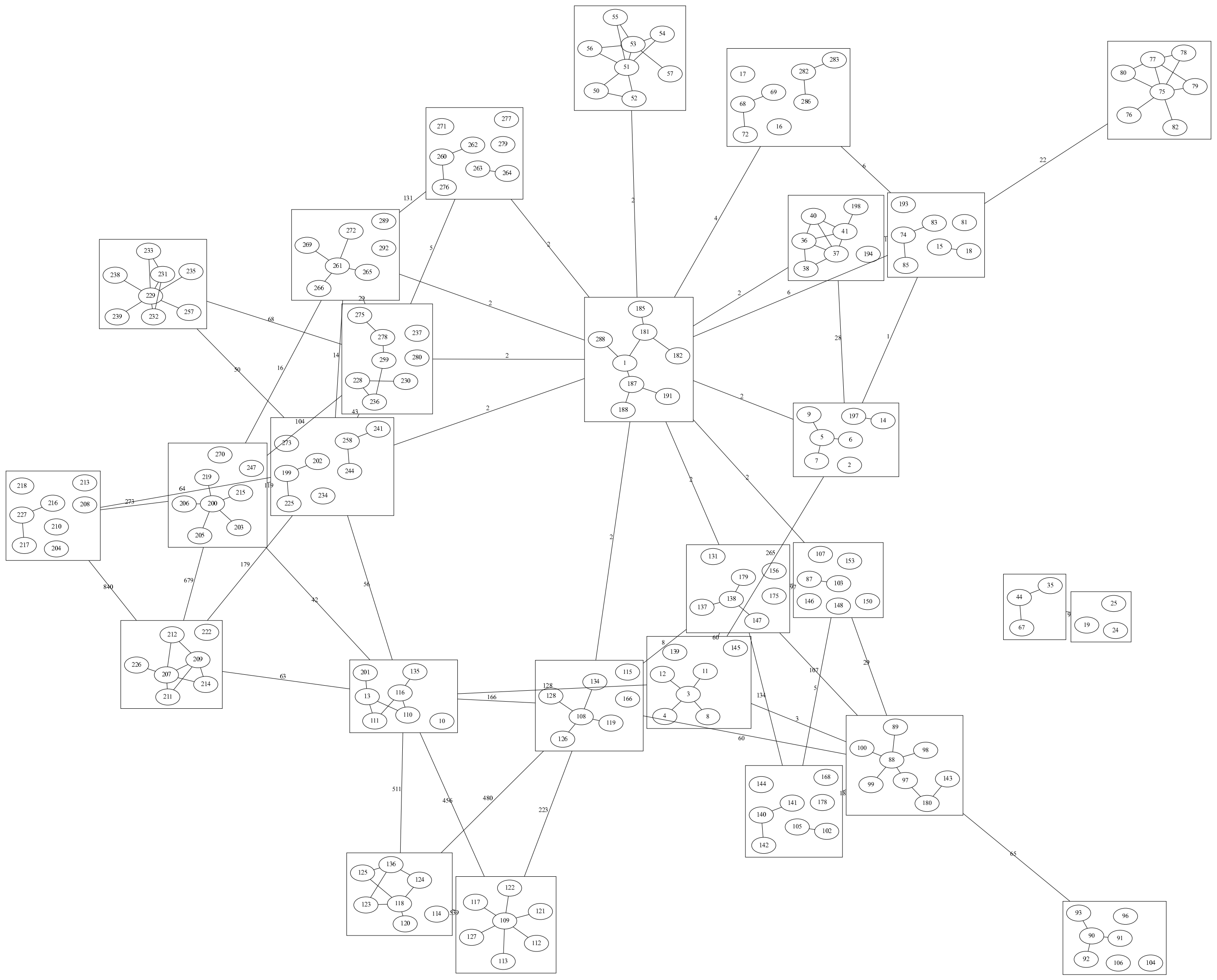


Figure 6.9. Partitioned traffic graph of the Cockroach agent.

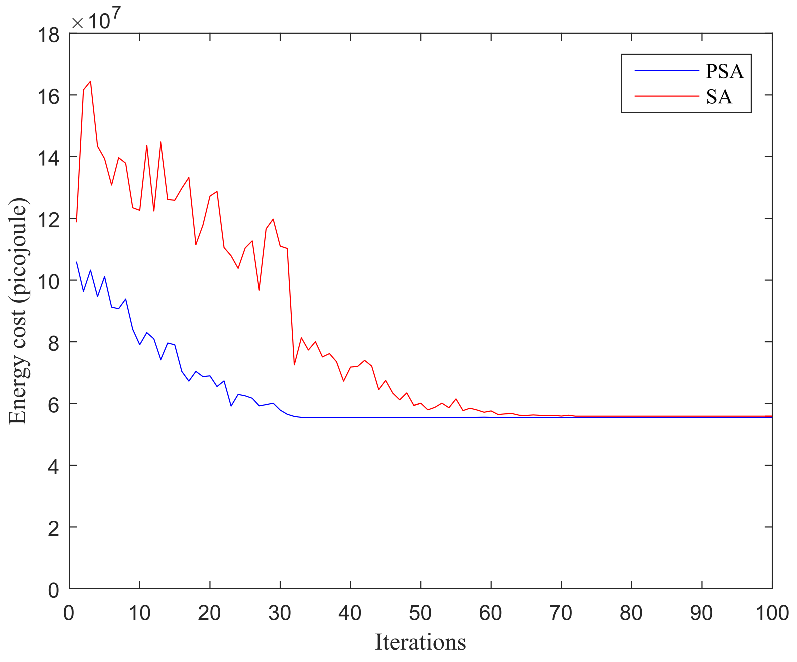


Figure 6.10. Energy cost comparison between PSA and SA in each iteration for the Cockroach agent.

To further investigate the performance of the proposed mapping technique, a set of 10 traffic graphs were synthetically created for numbers of GPEs ranging from 4 to 196. These GPEs were mapped onto the network using PSA and SA techniques. Figure 6.11 illustrates the estimated energy cost comparison of the mapping techniques. It shows that the cost was almost identical for both approaches. However when the computation times were compared, as shown in Figure 6.12, PSA provided a significant improvement over SA for higher number GPEs. This justifies the efficiency of the PSA technique for large systems in terms of computation time without sacrificing mapping quality.

6.6 Conclusion

In NoCs design, determining a mapping of a large numbers of PEs to routers in a reasonably short period of time is one of the most challenging tasks. This chapter proposes a NoC mapping technique for systems that typically contain hundreds of homogeneous PEs. The number of routers is reduced by using network partitioning to divide the PE traffic graph into a graph containing a smaller number of GPEs. This also aids in reducing the number of links in the network structure. Multi-level graph

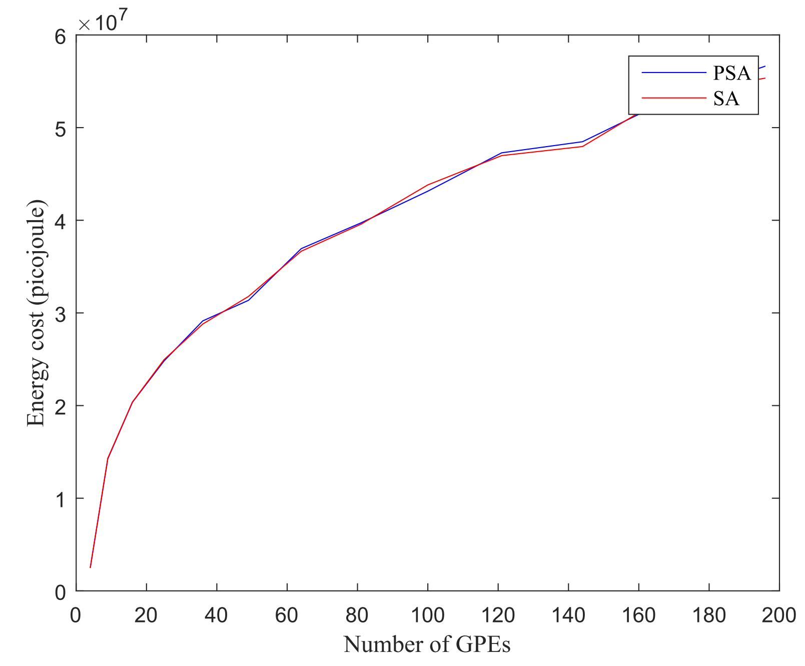


Figure 6.11. Energy cost comparison between PSA and SA.

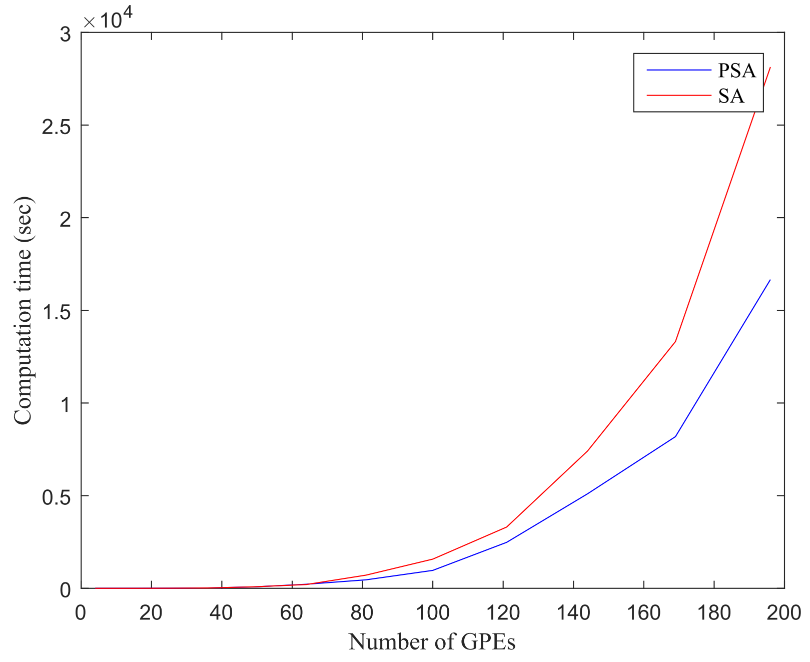


Figure 6.12. Computation time comparison between PSA and SA.

partitioning is used to produce a good quality partitioning in a small amount of time. This puts PEs that communicate a lot into the same GPE under a single router; therefore the traffic over the network is reduced. Furthermore, partitioning significantly reduces the search space of the mapping problem, as instead of mapping individual PEs, one GPE is mapped to each router. This reduces the computation time of mapping process.

A priority-based algorithm is proposed in this chapter to map PEs to NoC routers. This is a refinement of the SA algorithm using localised swapping in the annealing stages. The proposed technique is validated using two test cases as well as a set of synthetically created traffic graphs. The experiments demonstrate that all of these optimisation techniques make the PE-to-router mapping much more time efficient without compromising the mapping quality, which is measured by evaluating the energy costs of the mapping solutions.

The results show that for the test cases of the self-configurable agent and Cockroach agent, the proposed PSA mapping works at least two times faster than original SA mapping while producing equally good solutions. Although it does not run as fast as BB mapping, the computation time difference between BB and PSA is significantly less than that between BB and SA. We believe that a better initial mapping will reduce the difference even further. Compared to BB, Simulated Annealing based techniques will generally provide better memory utilisation, as discussed in the last chapter. Considering this along with the computation time improvement over SA, the PSA is the best candidate amongst the three, for NoC mapping in large systems.

The performance improvement is compared with the initial mapping. No experiment has been done in this work to compare performance improvement after subsequent regrouping. Further work to be done includes testing the hypothesis that the re-mapped solutions perform better (by running faster and producing solutions that consume less energy) than previous mapping configurations. Also, there is a need to examine how a better initial mapping affects the overall mapping performance and system performance, as a whole. Although this priority-based mapping is conducted on the Street agents, it is general enough to be used in other homogeneous systems as well.

Chapter 7

Conclusions and Future Work

THIS chapter summarises the research presented in this thesis, and gives comments on the significance of the work in the field of production systems and networks on chip. It highlights the contributions that the research has made towards designing an optimisation algorithm for mapping homogeneous processing elements of a system on chip onto a NoC structure. The chapter also identifies areas for future work to further advance the research on human-like intelligent systems and on-chip communication.

7.1 Summary

This thesis has reported an investigation of the communication platform of a hardware-based production system, which was built as a SoC having a large number of homogeneous PEs. As on-chip communication is generally a crucial performance-determining factor in a large scale multi-processor system, NoCs were argued to be a communication architecture that could be systematically tailored to address this challenge. The mapping of PEs to NoC routers, which significantly influences the performance of overall system, was addressed as the major issue to be examined in this thesis.

Chapter 2 presented a background study and review of related works to establish the basis for the work described in the thesis. It explained the significance of cognitive architectures in AGI research. Two successful cognitive architectures, Soar and ACT-R, were discussed to demonstrate examples of the use of production systems in cognitive systems. This chapter reported different approaches to realising production systems. It was explained that conventional computers, in spite of their impressive performance improvement over several decades, do not provide an efficient platform for such applications. In contrast, SoCs with many PEs were proposed as a technology suitable for building a hardware-based production systems. To provide an efficient communication between the PEs, a NoC-based interconnect platform was suggested as the most suitable approach. This chapter pointed out the importance of mapping of PEs to NoC routers, and reviewed the research activities in this field.

A new hardware-based production system, the Street Processor, was discussed in Chapter 3. It reviewed many aspects and characteristics of the hardware. The processor has its own instruction set, called Street language, which was inspired by OPS5 and the languages used in Soar and ACT-R, but is much simpler than them. The Street Processor supports fine-grained parallelism by implementing the execution of production rules in simple, homogeneous PEs, and distributing computation and memory among them. The PEs include customised CAM to allow parallel search operations to find new rule instantiations. The Street Processor introduced the concept of a periodic sleep interval, during which the execution is paused, and the placement of the PEs on NoC structure is updated, to improve overall traffic, latency and power consumption. The processor also allows its memory to expand over multiple PEs to avoid memory overflow when it executes for an extended period of time.

Chapter 4 described a NoC-based communication platform for SoCs with a large number of homogeneous PEs. The Street Processor is an example of this class of SoCs.

The NoC architecture used a regular mesh topology, which was considered to be the best fit for a homogeneous system. A dimension order XY routing was used in this NoC design because of its deadlock- and livelock-freedom, and low buffering requirement. The architecture employed wormhole routing and virtual channels as flow control strategies. Wormhole routing was selected because of its limited buffering requirement and reduced latency. Virtual channels improved the efficiency of data traversal by reducing the head-of-line blocking problem. This chapter also discussed the micro-architecture of a NoC router addressing the above mentioned requirements. Unlike most NoC routers, which typically support one PE each, this router was connected to multiple PEs to reduce the system implementation cost. The interfaces of the router used small static FIFO buffers to store the flits. Furthermore, the router used credit-based link level flow control to coordinate flit delivery between routers by keeping track of the number of buffers available at the adjacent router.

The mapping of PEs to NoC routers is an important performance-determining factor of SoC architectures because the communication time, required link bandwidth and power consumption are dependent on the placements of PEs. But optimum mapping of PEs to routers is an instance of NP-hard optimisation problems, in which the search space of the problem increases factorially with the system size. Chapter 5 explored two optimisation algorithms, Branch and Bound and Simulated Annealing to map the homogeneous PEs of the Street Processor to NoC routers. To compare the performance of these mapping techniques, a self-configurable agent was implemented in Street language using a total of 70 production rules. The mapping techniques were employed to map the PEs representing the production rules onto a 9×9 mesh structure. Although the solution qualities of both of the techniques were almost identical in terms of their energy costs, the Branch and Bound based mapping technique executed approximately 5 times faster than the Simulated Annealing based technique. This was because Simulated Annealing was a blind algorithm, which wasted much time exploring unpromising solutions, whereas Branch and Bound systematically deleted almost 95% of its solution space without completely exploring it. However, Branch and Bound achieved this computation time advantage at the cost of a high memory requirement. For applications like the Street Processor, for which memory optimisation is more important than computation time, a Simulated Annealing based technique was considered to be a more suitable solution.

In Chapter 6, a multi-level graph partitioning technique was applied to the PE traffic graph to split it into smaller groups of PEs in order to assign each GPE to a router. This particular graph partitioning technique provided a balanced traffic distribution among the GPEs in a short period of time compared to BB and SA. It placed heavily communicating PEs into the same GPE under a single router. This reduced the overall traffic over the network since the local PEs communicated to each other internally without using network links. This chapter also described an improved Simulated Annealing-based mapping technique. It analysed the GPE dependencies and router positions to prioritise them. These priorities were used to determine a heuristic initial mapping. The proposed technique also involved a localised swapping strategy, which selectively allowed GPEs to be swapped between routers having the same or different priorities depending on the annealing stages. These efforts resulted in faster convergence to a final mapping solution. This Priority-based Simulated Annealing was applied to two test cases as well as a set of synthetically created traffic graphs to map the PEs to NoC routers. Experiments showed that compared to the original Simulated Annealing algorithm, PSA performed at least two times faster while producing equally good solutions, as indicated by the estimated energy cost consumption of the mapped solutions. Although this proposed mapping technique was not as fast as Branch and Bound-based mapping, it was still preferred among the three techniques considered, because of the high memory requirement of Branch and Bound algorithm and the quality of the solution.

7.2 Conclusions

The thesis has provided contributions in defining an interconnect platform, particularly the mapping optimisation algorithm, for SoCs having a large number of PEs. The Street Processor may include thousands of homogeneous PEs executing production rules. To make the communication between this very large number of PEs efficient, a NoC-based interconnect architecture has been discussed in this thesis. Although NoCs are not a very new concept in the field of SoCs, the application of NoCs to such a large scale network of fine-grained homogeneous PEs can be considered very promising.

The mapping of PEs to NoC routers directly impacts the performance of a SoC. This instance of NP-hard problem cannot be solved by deterministic approaches. This work has inspected Branch and Bound, and Simulated Annealing optimisation algorithms to solve the mapping problem. For the systems like the Street Processor that concern more about optimising memory, but have the ability to periodically re-organise its PEs, Simulated Annealing based mapping has been advocated to be more suitable. The proposed priority-based Simulated Annealing algorithm with a heuristic initial mapping and localised swapping has ended up in faster computation to find an optimum mapping solution. We have used the proposed technique for Street agents of approximately three hundred production rules. However, we consider that this number is still not enough for complex agents. The performance of NoC mapping techniques for agents containing thousands of PEs are not yet evaluated because of the lack of availability to date of such large agents.

In this thesis, we have considered generic NoC routers, which supports multiple PEs. There still exists potential for improvement of the router architecture. The trade-off between the number of routers and the number of interfaces per router to support the same number of PEs should be investigated. Moreover, this work included some efforts towards building a self-manageable cognitive agent. To assist this goal, a self-configurable agent was realised using production rules written in Street language. The objective of this work is to demonstrate the ability of Street language to solve the PE mapping problem, and to develop a test case to measure the performance of the proposed mapping algorithm. Further work needs to be done to define the procedure of updating rule definition of the PEs, synchronisation techniques and other related issues.

Overall, the thesis has advanced the research of hardware-based architectures for AI and cognitive systems by investigating issues in the underlying communication platform. Although the proposed mapping technique has been tested for PEs of the Street Processor, we believe that this is general enough to be applied to any large scale homogeneous SoCs as well.

7.3 Future work

In this thesis, we explored the communication platform of an on-chip production system. Inevitably, some topics and ideas arose in the study, which could not be investigated due to time constraints. Some of them are listed below.

- The performance of the mapping techniques were examined using several Street agents having a maximum of 293 production rules. All of them were software agents executed within a controlled environment. There is a need for bigger and more complex hardware agents to provide more realistic results to demonstrate the suitability of the work.

- This thesis mainly concentrated on the mapping of homogeneous PEs onto NoC platform. The issues related to the design of multi-interface NoC router architecture were not investigated in detail in this research. The trade-off between attaching PEs to router interfaces instead of individual routers was not worked out here. The PEs under the same router communicated using the crossbar of the router. A number of alternative approaches ranging from point-to-point connection to a hierarchy of NoC can be explored for better solutions. Also, the impact of buffer size and flit size on overall NoC performance and complexity may also be an interesting topic for future research.

- A self-configurable Street agent was introduced to demonstrate the ability of Street language to solve NoC mapping problems, and to build a test case to evaluate the performance of mapping algorithms. However, details about the mechanism of updating PE definitions, required synchronisation techniques and other related issues were not explored in this thesis. In order to build an operational self-configurable agent, these issues should be investigated.

- The mapping algorithms adopted the energy model considered in the experiments of (Hu and Marculescu 2005), where the energy consumptions were measured with Synopsys design compiler for a 0.35 μm technology. Since this energy model was used for comparison purpose, we considered it acceptable. However, to get accurate energy costs of different mapping approaches, a more sophisticated energy model of homogeneous system supported by experimental results, is required.

- The performance improvement of the proposed priority-based simulated annealing technique was compared with the initial mapping. No experiment was done to compare it after subsequent re-grouping. A further investigation on performance comparison between subsequent mapping solutions may reveal interesting information.

- The proposed mapping technique was intended for implementation at physical and architectural level, but currently limited at functional level. An obvious direction for further work is to complement this with hardware prototypes, probably using FPGAs. Such prototypes will bring forward many interesting issues, and will certainly encourage the researchers to mitigate the issues.

Appendix A

THIS appendix lists the names of Street production rules used to realise a self-configurable agent. This re-organises the processing elements on a NoC structure using the Branch and Bound algorithm. The complete definition of the rules are available in the website of the Street project (Phillips 2016).

Table A.1. List of Street production rules of the self-configurable agent.

PE index	*Street* production rule
0	initialise
1	calculateTotalPE
2	calculateIntermediateValuesForTotalLinks
3	calculateTotalEdges
4	calculateTotalLinks
5	initialisePEandTiles
6	findRowTimesColumns
7	findColumn
8	initialiseAdjacency
9	calculateRowColDifference
10	getPositiveRowDiff
11	getPositiveColumnDiff
12	calculateDistance
13	initialiseTrafficRandom
14	ackTrafficRandom
15	checkAllLoadGenerated
16	checkAllDistanceCalculated
17	calculatePEtoPEtraffic
18	calculateTotalLoad
19	getAllRelations
20	createFirstMappingNodes
21	createNewNodes
22	getTotalRelationInNode
23	getPreviousRelations
24	getUnmappedPE
25	getUnoccupiedTile
26	getPartialTileToTileCost
27	getPartialMappingCost
28	flagCurrentCostCalculated
29	initialiseUnmappedPEandTile
30	initialiseLastUnmappedPEandTile
31	getMinMUminusCurrentDistance
32	getMinMUtileDistance
33	getTotalMUtraffic
34	getTotalNumberOfMUtraffic
35	flagTotalMUtrafficCalculated
36	getTotalMUcost
37	getMinUUminusCurrentDistance
38	getMinUUtileDistance
39	getUUtraffic
40	getTotalUUtraffic

PE index	***Street* production rule**
41	getTotalUUcost
42	getTotalMUandUUcostforLBcost
43	getLowerBoundCost
44	getTotalMUtrafficTimesRowAndCol
45	getSumRow
46	getSumCol
47	getGoodRowAndCol
48	getDistanceBetweenGoodAndU_tile
49	getGreedyRelationsAndMinDistance
50	getGreedyMUtileToTileCost
51	getGreedyMUcost
52	flagGreedyMUcostCalculated
53	getGreedyUUtileToTileCost
54	getGreedyUUcost
55	flagGreedyUUcostCalculated
56	flagUUcostCalculated_forOneUnmappedPE
57	getGreedyMUandUUcost
58	getUpperBoundCost
59	getMinMinusUBCost
60	getMinUBCost
61	deletePositiveMinMinusUBCost
62	flagMinUBCostCalculated
63	getLBminusMinUBcost
64	flagNodeExpandable
65	flagNodeExpandable_false
66	createMapping
67	getBestMappingCostDiff
68	deleteHigherMappingCostDiff
69	getBestMappingCost

Appendix B

THIS appendix lists the names of the Street production rules used for the development of the Cockroach agent. This agent was developed by James Donovan, an undergraduate student at the University of Adelaide. The work is reported in detail in the website of the Street Project (Phillips 2016).

Table B.1. List of Street production rules of the Cockroach agent.

PE index	*Street* production rule
0	initialise
1	Forward_init
2	Forward_stateRegister
3	Forward_nextstate_NIL_CHECK
4	Forward_output_NIL_CHECK
5	Forward_nextstate_CHECK_NIL
6	Forward_output_CHECK_NIL
7	Forward_nextstate_CHECK_FORWARD
8	Forward_output_CHECK_FORWARD
9	Forward_nextstate_FORWARD_OUT
10	Forward_output_FORWARD_OUT
11	Forward_nextstate_OUT_NIL
12	Forward_output_OUT_NIL
13	Forward_currentBusy
14	RandomAngleEvent_init
15	RandomAngleEvent_stateRegister
16	RandomAngleEvent_nextstate_NIL_HUGGINGSTART_left
17	RandomAngleEvent_nextstate_NIL_HUGGINGSTART_right
18	RandomAngleEvent_output_NIL_HUGGINGSTART
19	RandomAngleEvent_nextstate_HUGGINGSTART_HUGGINGSETTIMER
20	RandomAngleEvent_output_HUGGINGSTART_HUGGINGSETTIMER
21	RandomAngleEvent_nextstate_HUGGINGSETTIMER_HUGGINGWAIT
22	RandomAngleEvent_output_HUGGINGSETTIMER_HUGGINGWAIT
23	RandomAngleEvent_nextstate_HUGGINGWAIT_NIL
24	RandomAngleEvent_nextstate_HUGGINGWAIT_NIL_nowall
25	RandomAngleEvent_nextstate_HUGGINGWAIT_TURNLEFT
26	RandomAngleEvent_nextstate_HUGGINGWAIT_TURNRIGHT
27	RandomAngleEvent_output_HUGGINGWAIT_nextstate
28	RandomAngleEvent_nextstate_TURNRIGHT_WAIT
29	RandomAngleEvent_output_TURNRIGHT_WAIT
30	RandomAngleEvent_nextstate_TURNLEFT_WAIT
31	RandomAngleEvent_output_TURNLEFT_WAIT
32	RandomAngleEvent_nextstate_WAIT_NIL
33	RandomAngleEvent_output_WAIT_NIL
34	RandomAngleEvent_currentWallStatus
35	Turn_init
36	Turn_stateRegister

PE index	*Street* production rule
37	Turn_nextstate_NIL_ROTATE
38	Turn_output_NIL_ROTATE
39	Turn_nextstate_NIL_NIL
40	Turn_output_NIL_NIL
41	Turn_nextstate_ROTATE_NIL
42	Turn_output_ROTATE_NILL
43	WallDetector_nowall
44	WallDetector_leftwall
45	WallDetector_rotateright
46	WallDetector_rightwall
47	WallDetector_rotateleft
48	WallDetector_rotaterandom
49	WallHugger_init
50	WallHugger_stateRegister
51	WallHugger_nextstate_NIL_WAIT
52	WallHugger_output_NIL_WAIT
53	WallHugger_nextstate_WAIT_OUTLEFT
54	WallHugger_nextstate_WAIT_OUTRIGHT
55	WallHugger_nextstate_WAIT_OUTRANDOM
56	WallHugger_output_WAIT_nextstate
57	WallHugger_nextstate_OUTLEFT_NIL
58	WallHugger_output_OUTLEFT_NIL
59	WallHugger_nextstate_OUTRIGHT_NIL
60	WallHugger_output_OUTRIGHT_NIL
61	WallHugger_nextstate_OUTRANDOM_OUTRANDOM1
62	WallHugger_output_OUTRANDOM_OUTRANDOM1
63	WallHugger_nextstate_OUTRANDOM1_NIL
64	WallHugger_output_OUTRANDOM1_NIL_0
65	WallHugger_nextstate_OUTRANDOM1_NIL_1
66	WallHugger_currentWallStatus
67	TurnSuppressor_init
68	TurnSuppressor_inhibitionCount
69	TurnSuppressor_inhibitionMessageArrived
70	TurnSuppressor_inhibitionPeriodOver
71	TurnSuppressor_passMessage
72	TurnSuppressor_inhibitMessage
73	Frighteness_init
74	Frighteness_stateRegister

PE index	***Street* production rule**
75	Frightness_nextstate_NIL_CHECK
76	Frightness_output_NIL_CHECK
77	Frightness_nextstate_CHECK_CONSUME
78	Frightness_nextstate_CHECK_RESTORE
79	Frightness_output_CHECK_nextstate
80	Frightness_nextstate_RESTORE_NIL
81	Frightness_output_RESTORE_NIL
82	Frightness_output_RESTORE_NIL_limited
83	Frightness_nextstate_CONSUME_NIL
84	Frightness_output_CONSUME_NIL
85	Frightness_output_CONSUME_NIL_limited
86	Tiredness_init
87	Tiredness_stateRegister
88	Tiredness_nextstate_NIL_CHECKDARK
89	Tiredness_output_NIL_CHECKDARK
90	Tiredness_nextstate_CHECKDARK_OUTREST
91	Tiredness_nextstate_CHECKDARK_OUTDARK
92	Tiredness_output_CHECKDARK_nextstate
93	Tiredness_nextstate_OUTRESET_OUTDARK
94	Tiredness_output_OUTRESET_OUTDARK
95	Tiredness_nextstate_OUTDARK_CHECK
96	Tiredness_output_OUTDARK_CHECK
97	Tiredness_nextstate_CHECK_RESTORE
98	Tiredness_nextstate_CHECK_CONSUME
99	Tiredness_nextstate_CHECK_nextstate
100	Tiredness_nextstate_RESTORE_NIL
101	Tiredness_output_RESTORE_NIL
102	Tiredness_output_RESTORE_NIL_limited
103	Tiredness_nextstate_CONSUME_NIL
104	Tiredness_output_CONSUME_NIL
105	Tiredness_state_CONSUME_NIL_limited
106	Tiredness_currentResting
107	IntegrateDark_init
108	IntegrateDark_stateRegister
109	IntegrateDark_nextstate_NIL_CHECKRESET
110	IntegrateDark_output_NIL_CHECKRESET
111	IntegrateDark_nextstate_CHECKRESET_SETINCR
112	IntegrateDark_output_CHECKRESET_SETINCR_true
113	IntegrateDark_output_CHECKRESET_SETINCR_false
114	IntegrateDark_nextstate_SETINCR_SETPOS

PE index	*Street* production rule
115	IntegrateDark_output_SETINCR_SETPOS
116	IntegrateDark_nextstate_SETPOS_OUT
117	IntegrateDark_output_SETPOS_OUT_up_1
118	IntegrateDark_output_SETPOS_OUT_up_2
119	IntegrateDark_output_SETPOS_OUT_right
120	IntegrateDark_output_SETPOS_OUT_down
121	IntegrateDark_output_SETPOS_OUT_left
122	IntegrateDark_output_SETPOS_OUT_topright
123	IntegrateDark_output_SETPOS_OUT_bottomright
124	IntegrateDark_output_SETPOS_OUT_bottomleft
125	IntegrateDark_output_SETPOS_OUT_topleft
126	IntegrateDark_nextstate_OUT_NIL
127	IntegrateDark_output_OUT_NIL
128	IntegrateDark_currentHeading
129	IntegrateDark_resetRequest
130	IntegrateDark_timerCount
131	IntegrateDark_timerExpired
132	IntegrateDark_sendFoundDarkTrue
133	IntegrateDark_sendFoundDarkFalse
134	IntegrateDark_incrDistanceDiagonalTemp
135	IntegrateDark_incrDistanceDiagonal
136	PathPlanDark_init
137	PathPlanDark_stateRegister
138	PathPlanDark_nextstate_NIL_WAIT
139	PathPlanDark_output_NIL_WAIT
140	PathPlanDark_nextstate_WAIT_CHECK1
141	PathPlanDark_output_WAIT_CHECK1
142	PathPlanDark_nextstate_CHECK1_SLEEP1_a
143	PathPlanDark_nextstate_CHECK1_SLEEP1_b
144	PathPlanDark_nextstate_CHECK1_CHECK2_a
145	PathPlanDark_nextstate_CHECK1_CHECK2_b
146	PathPlanDark_nextstate_CHECK1_ABANDON
147	PathPlanDark_output_CHECK1_nextstate
148	PathPlanDark_nextstate_CHECK2_SLEEPEND
149	PathPlanDark_nextstate_CHECK2_FRIGHT1
150	PathPlanDark_output_CHECK2_nextstate
151	PathPlanDark_nextstate_SLEEP1_SLEEP3
152	PathPlanDark_nextstate_SLEEP1_SLEEP2
153	PathPlanDark_output_SLEEP1_nextstate
154	PathPlanDark_nextstate_SLEEP2_SLEEP2b

PE index	*Street* production rule
155	PathPlanDark_output_SLEEP2_SLEEP2b
156	PathPlanDark_nextstate_SLEEP2b_SLEEP3
157	PathPlanDark_output_SLEEP2b_SLEEP3
158	PathPlanDark_nextstate_SLEEP3_SLEEP4
159	PathPlanDark_nextstate_SLEEP3_SLEEPEND
160	PathPlanDark_output_SLEEP3_nextstate
161	PathPlanDark_nextstate_SLEEP4_NIL
162	PathPlanDark_output_SLEEP4_NIL
163	PathPlanDark_nextstate_SLEEPEND_SLEEPEND2
164	PathPlanDark_output_SLEEPEND_SLEEPEND2
165	PathPlanDark_nextstate_SLEEPEND2_NIL_a
166	PathPlanDark_nextstate_SLEEPEND2_NIL_b
167	PathPlanDark_nextstate_SLEEPEND2_FRIGHT1_a
168	PathPlanDark_nextstate_SLEEPEND2_FRIGHT1_b
169	PathPlanDark_output_SLEEPEND2_nextstate
170	PathPlanDark_nextstate_FRIGHT1_FRIGHT2
171	PathPlanDark_nextstate_FRIGHT1_NIL
172	PathPlanDark_output_FRIGHT1_nextstate
173	PathPlanDark_nextstate_FRIGHT2_FRIGHT2b
174	PathPlanDark_output_FRIGHT2_FRIGHT2b
175	PathPlanDark_nextstate_FRIGHT2b_NIL
176	PathPlanDark_output_FRIGHT2b_nextstate
177	PathPlanDark_nextstate_ABANDON_NIL
178	PathPlanDark_output_ABANDON_NIL
179	PathPlanDark_currentInputLevel
180	RandomAngleEventInhibitor_init
181	RandomAngleEventInhibitor_inhibitionCount
182	RandomAngleEventInhibitor_inhibitionMessageArrived
183	RandomAngleEventInhibitor_inhibitionPeriodOver
184	RandomAngleEventInhibitor_passMessage
185	RandomAngleEventInhibitor_inhibitMessage
186	WallHuggerInhibitor_init
187	WallHuggerInhibitor_inhibitionCount
188	WallHuggerInhibitor_inhibitionMessageArrived
189	WallHuggerInhibitor_inhibitionPeriodOver
190	WallHuggerInhibitor_passMessage
191	WallHuggerInhibitor_inhibitMessage
192	ForwardSuppressor_init
193	ForwardSuppressor_inhibitionCount
194	ForwardSuppressor_inhibitionMessageArrived

PE index	***Street* production rule**
195	ForwardSuppressor_inhibitionPeriodOver
196	ForwardSuppressor_passMessage
197	ForwardSuppressor_inhibitMessage
198	IntegrateFood_init
199	IntegrateFood_stateRegister
200	IntegrateFood_nextstate_NIL_CHECKRESET
201	IntegrateFood_output_NIL_CHECKRESET
202	IntegrateFood_nextstate_CHECKRESET_SETINCR
203	IntegrateFood_output_CHECKRESET_SETINCR_true
204	IntegrateFood_output_CHECKRESET_SETINCR_false
205	IntegrateFood_nextstate_SETINCR_SETPOS
206	IntegrateFood_output_SETINCR_SETPOS
207	IntegrateFood_nextstate_SETPOS_OUT
208	IntegrateFood_output_SETPOS_OUT_up_1
209	IntegrateFood_output_SETPOS_OUT_up_2
210	IntegrateFood_output_SETPOS_OUT_right
211	IntegrateFood_output_SETPOS_OUT_down
212	IntegrateFood_output_SETPOS_OUT_left
213	IntegrateFood_output_SETPOS_OUT_topright
214	IntegrateFood_output_SETPOS_OUT_bottomright
215	IntegrateFood_output_SETPOS_OUT_bottomleft
216	IntegrateFood_output_SETPOS_OUT_topleft
217	IntegrateFood_nextstate_OUT_NIL
218	IntegrateFood_output_OUT_NIL
219	IntegrateFood_currentHeading
220	IntegrateFood_resetRequest
221	IntegrateFood_timerCount
222	IntegrateFood_timerExpired
223	IntegrateFood_sendFoundFoodTrue
224	IntegrateFood_sendFoundFoodFalse
225	IntegrateFood_incrDistanceDiagonalTemp
226	IntegrateFood_incrDistanceDiagonal
227	PathPlanFood_init
228	PathPlanFood_stateRegister
229	PathPlanFood_nextstate_NIL_WAIT
230	PathPlanFood_output_NIL_WAIT
231	PathPlanFood_nextstate_WAIT_CHECK1
232	PathPlanFood_output_WAIT_CHECK1
233	PathPlanFood_nextstate_CHECK1_EAT1_a
234	PathPlanFood_nextstate_CHECK1_EAT1_b

PE index	*Street* production rule
235	PathPlanFood_nextstate_CHECK1_CHECK2_a
236	PathPlanFood_nextstate_CHECK1_CHECK2_b
237	PathPlanFood_nextstate_CHECK1_ABANDON
238	PathPlanFood_output_CHECK1_nextstate
239	PathPlanFood_nextstate_CHECK2_EATEND
240	PathPlanFood_nextstate_CHECK2_NIL
241	PathPlanFood_output_CHECK2_nextstate
242	PathPlanFood_nextstate_EAT1_EAT3
243	PathPlanFood_nextstate_EAT1_EAT2
244	PathPlanFood_output_EAT1_nextstate
245	PathPlanFood_nextstate_EAT2_EAT2b
246	PathPlanFood_output_EAT2_EAT2b
247	PathPlanFood_nextstate_EAT2b_EAT3
248	PathPlanFood_output_EAT2b_EAT3
249	PathPlanFood_nextstate_EAT3_EAT4
250	PathPlanFood_nextstate_EAT3_EATEND
251	PathPlanFood_output_EAT3_nextstate
252	PathPlanFood_nextstate_EAT4_NIL
253	PathPlanFood_output_EAT4_NIL
254	PathPlanFood_nextstate_EATEND_NIL
255	PathPlanFood_output_EATEND_NIL
256	PathPlanFood_nextstate_ABANDON_NIL
257	PathPlanFood_output_ABANDON_NIL
258	PathPlanFood_currentInputLevel
259	Hunger_init
260	Hunger_stateRegister
261	Hunger_nextstate_NIL_CHECKFOOD
262	Hunger_output_NIL_CHECKFOOD
263	Hunger_nextstate_CHECKFOOD_OUTRESET
264	Hunger_nextstate_CHECKFOOD_OUTFOOD
265	Hunger_output_CHECKFOOD_nextstate
266	Hunger_nextstate_OUTRESET_OUTFOOD
267	Hunger_output_OUTRESET_OUTFOOD
268	Hunger_nextstate_OUTFOOD_CHECK
269	Hunger_output_OUTFOOD_CHECK
270	Hunger_nextstate_CHECK_RESTORE
271	Hunger_nextstate_CHECK_CONSUME
272	Hunger_nextstate_CHECK_nextstate
273	Hunger_nextstate_RESTORE_NIL
274	Hunger_output_RESTORE_NIL

PE index	***Street* production rule**
275	Hunger_output_RESTORE_NIL_limited
276	Hunger_nextstate_CONSUME_NIL
277	Hunger_output_CONSUME_NIL
278	Hunger_state_CONSUME_NIL_limited
279	Hunger_currentEating
280	Hunger_foundFood
281	PathPlanDarkAngleInhibitor_init
282	PathPlanDarkAngleInhibitor_inhibitionCount
283	PathPlanDarkAngleInhibitor_inhibitionMessageArrived
284	PathPlanDarkAngleInhibitor_inhibitionPeriodOver
285	PathPlanDarkAngleInhibitor_passMessage
286	PathPlanDarkAngleInhibitor_inhibitMessage
287	PathPlanDarkRestingInhibitor_init
288	PathPlanDarkRestingInhibitor_inhibitionCount
289	PathPlanDarkRestingInhibitor_inhibitionMessageArrived
290	PathPlanDarkRestingInhibitor_inhibitionPeriodOver
291	PathPlanDarkRestingInhibitor_passMessage
292	PathPlanDarkRestingInhibitor_inhibitMessage

Bibliography

Aarts, E.., and Lenstra, J. K.. (eds.) (1997). *Local Search in Combinatorial Optimization*, 1st edn, John Wiley & Sons, Inc., New York, NY, USA.

ACKLAND-B., ANESKO-A., BRINTHAUPT-D., DAUBERT-S. J., KALAVADE-A., KNOBLOCH-J., MICCA-E., MOTURI-M., NICOL-C. J., AND O'NEILL-J. H. (2000). A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP, *IEEE Journal of Solid-state Circuits*, **35**(3), pp. 412–424.

AGARWAL-A., ISKANDER-C., AND SHANKAR-R. (2009). Survey of network on chip (NOC) architectures & contributions, *Journal of Engineering, Computing and Architecture*, **3**(1), pp. 21–27.

AMARAL-J. N., AND GHOSH-J. (1993). *Speeding Up Production Systems: From Concurrent Matching to Parallel Rule Firing*, pp. 139–160.

ANDERSON-J., AND LEBIERE-C. (2003). The Newell test for a theory of cognition, *Behavioral and Brain Science*, **26**, pp. 587–637.

ANDERSON-J. R. (1983). *The Architecture of Cognition*, Harvard University Press, Cambridge, MA.

ANDERSON-J. R. (1996). ACT: A simple theory of complex cognition, *American Psychologist*, **51**(4), p. 355.

ANDERSON-J. R. (2007). *How can the human mind exist in the physical universe?*, Oxford University Press, New York.

ANDERSON-J. R., AND BOWER-G. H. (1973). *Human associative memory*, Winston and Sons, Washington, DC.

ANDERSON-J. R., BOTHELL-D., BYRNE-M. D., DOUGLASS-S., LEBIERE-C., AND QIN-Y. (2004). An integrated theory of the mind, *Psychological Review*, **111**(4), pp. 1036–1060.

ANGELOV-P. P. (2013). *Evolving rule-based models: a tool for design of flexible adaptive systems*, Vol. 92, Physica.

AZULSYSTEMS. (2016). Vega processor.

BACKUS-J. (1978). Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs, *Communications of the ACM*, **21**, pp. 613–641.

BADER-D. A., HART-W. E., AND PHILLIPS-C. A. (2005). *Parallel Algorithm Design for Branch and Bound*, Springer New York, New York, NY, pp. 5–1–5–44.

BARNARD-S. T., AND SIMON-H. D. (1993). A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, *Proceedings of the sixth SIAM conference on Parallel Processing for Scientific Computing*, p. 711718.

BENINI-L., AND MICHELI-G. D. (2002). Networks on chips: A new SoC paradigm, *IEEE Computer*, **35**(1), pp. 70–78.

BHATTACHARYYA-S. S., DEPRETTERE-E. F., AND TEICH-J. (2003). *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*.

BJERREGAARD-T., AND MAHADEVAN-S. (2006). A Survey of Research and Practices of Network-on-chip, *ACM Comput. Surv.*, **38**(1), pp. 1–51.

BLACHNIK-M., AND DUCH-W. (2008). Building localized basis function networks using context dependent clustering, *International Conference on Artificial Neural Networks*, Springer, pp. 482–491.

BONA-A., ZACCARIA-V., AND ZAFALON-R. (2004). System level power modeling and simulation of high-end industrial network-on-chip, *Ultra Low-Power Electronics and Design*, Springer, pp. 233–254.

BOTHELL-D. (2004). ACT-R 6.1 Reference manual, *Technical report*, Tech. Rep., 2004.[Online]. Available: https://act-r. psy. cmu. edu/wordpress/wp-content/themes/ACTR/actr6. 1/reference-manual. pdf.

BROOKS-R. (1986). A robust layered control system for a mobile robot, *IEEE Journal on Robotics and Automation*, **2**(1), pp. 14–23.

BROOKS-R., AND CONNELL-J. (1986). *Asynchronous distributed control system for a mobile robot*, MIT Artificial Intelligence Lab.

BUDIU-R. (2013). ACT-R Homepage, http://act-r.psy.cmu.edu/.

BUDIU-R., AND ANDERSON-J. R. (2004). Interpretation-based processing: A unified theory of semantic sentence comprehension, *Cognitive Science*, **28**(1), pp. 1–44.

BYRNE-M. D., AND KIRLIK-A. (2005). Using computational cognitive modeling to diagnose possible sources of aviation error, *The international journal of aviation psychology*, **15**(2), pp. 135–155.

CAMPBELL-M., JR.-A. J. H., AND HSIUNG HSU-F. (2002). Deep Blue, *Artificial Intelligence*, **132**, pp. 57–83.

CARLONI-L. P., MCMILLAN-K. L., AND SANGIOVANNI-VINCENTELLI-A. L. (2001). Theory of latency-insensitive design, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **20**(9), pp. 1059–1076.

CARVALHO-E., MARCON-C., CALAZANS-N., AND MORAES-F. (2009). Evaluation of static and dynamic task mapping algorithms in noc-based mpsocs, *2009 International Symposium on System-on-Chip*, pp. 087–090.

CATONI-O. (1996). Metropolis, Simulated Annealing, and Iterated Energy Transformation Algorithms: Theory and Experiments, *Journal of Complexity*, **12**(4), pp. 595 – 623.

CHENG-C. K., AND WEI-Y. C. A. (1991). An improved two-way partitioning algorithm with stable performance VLSI, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **10**(12), pp. 1502–1511.

CHEN-Y., LIU-Z.-L., AND XIE-Y.-B. (2012). A knowledge-based framework for creative conceptual design of multi-disciplinary systems, *Computer-Aided Design*, **44**(2), pp. 146 – 153.

CHIEN-A. A., AND KIM-J. H. (1995). Planar-adaptive routing: low-cost adaptive networks for multiprocessors, *Journal of the ACM (JACM)*, **42**(1), pp. 91–123.

CHOU-C.-L., AND MARCULESCU-R. (2008). User-aware dynamic task allocation in networks-on-chip, *2008 Design, Automation and Test in Europe*, IEEE, pp. 1232–1237.

CHOU-C.-L., OGRAS-U. Y., AND MARCULESCU-R. (2008). Energy-and performance-aware incremental mapping for networks on chip with multiple voltage levels, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **27**(10), pp. 1866–1879.

CUTLER-K.-M. (2014). Artificial Intelligence Startup Vicarious Grabs Funding From Bezos, Benioff And Jerry Yang, http://techcrunch.com/2014/04/07/vicarious/.

DALL'OSSO-M., BICCARI-G., GIOVANNINI-L., BERTOZZI-D., AND BENINI-L. (2012). Xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs, *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, IEEE, pp. 45–48.

DALLY-W. J. (1992). Virtual-channel flow control, *IEEE Transactions on Parallel and Distributed systems*, **3**(2), pp. 194–205.

DALLY-W. J., AND AOKI-H. (1993). Deadlock-free adaptive routing in multicomputer networks using virtual channels, *IEEE transactions on Parallel and Distributed Systems*, **4**(4), pp. 466–475.

DALLY-W. J., AND SEITZ-C. L. (1987). Deadlock-free message routing in multiprocessor interconnection networks, *IEEE Transactions on computers*, **100**(5), pp. 547–553.

DALLY-W. J., AND TOWLES-B. (2001). Route Packets, Not Wires: On-chip Inteconnection Networks, *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, ACM, New York, NY, USA, pp. 684–689.

DALLY-W. J., AND TOWLES-B. P. (2004). *Principles and practices of interconnection networks*, Elsevier.

DE SOUZA CARVALHO-E. L., CALAZANS-N. L. V., AND MORAES-F. G. (2010). Dynamic task mapping for MPSoCs, *IEEE Design & Test of Computers*, **27**(5), pp. 26–35.

DUATO-J. (1995). A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks, *IEEE Transactions on Parallel and Distributed Systems*, **6**(10), pp. 1055–1067.

DUCH-W., ADAMCZAK-R., AND GRABCZEWSKI-K. (2001). A new methodology of extraction, optimization and application of crisp and fuzzy logical rules, *IEEE Transactions on Neural Networks*, **12**(2), pp. 277–306.

DUCH-W., OENTARYO-R. J., AND PASQUIER-M. (2008). Cognitive architectures: Where do we go from here?, *Proceedings of the 2008 Conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, IOS Press, Amsterdam, The Netherlands, The Netherlands, pp. 122–136.

DUTTA-S., JENSEN-R., AND RIECKMANN-A. (2001). Viper: A multiprocessor SoC for advanced set-top box and digital TV systems, *IEEE Design & Test of Computers*, **18**(5), pp. 21–31.

FANET-A. (2005). NoC: The arch key of IP integration methodology, *5th International Forum on Application-Specific Multi-Processor SoC*.

FEKR-A. R., KHADEMZADEH-A., JANIDARMIAN-M., AND BOKHARAEI-V. S. (2010). Bandwidth/fault tolerance/contention aware application-specific NoC using PSO as a mapping generator, *Proc. of The World Congress on Engineering*, pp. 247–252.

FERGUSON-G., AND ALLEN-J. F. (2011). A Cognitive Model for Collaborative Agents, *AAAI Fall Symposium: Advances in Cognitive Systems*, pp. 112–120.

FERRUCCI-D., BROWN-E., CHU-CARROLL-J., FAN-J., GONDEK-D., KALYANPUR-A. A., LALLY-A., MURDOCK-J. W., NYBERG-E., PRAGER-J., SCHLAEFER-N., AND WELTY-C. (2010). Building Watson: An overview of the DeepQA project, *AI Magazine*, **31**(3), pp. 59–79.

FIDUCCIA-C. M., AND MATTHEYSES-R. M. (1982). A linear-time heuristic for improving network partitions, *Design Automation, 1982. 19th Conference on*, pp. 175–181.

FLYNN-D. (1997). AMBA: enabling reusable on-chip designs, *IEEE micro*, **17**(4), pp. 20–27.

FORGY-C. (1979). *On the efficient implementation of production systems*, PhD thesis, Carnegie-Mellon University.

FORGY-C. L. (1981). OPS5 user's manual, *Technical report*, Computer Science Department, Carnegie-Mellon University.

FOUSKAKIS-D., AND DRAPER-D. (2002). Stochastic optimization: a review, *International Statistical Review*, **70**(3), pp. 315–349.

FRANKLIN-S. (2003). An autonomous software agent for navy personnel work: a case study, *In Human Interaction with Autonomous Systems in Complex Environments: Papers from 2003 AAAI Spring Symposium*.

FRANKLIN-S. (2007). A foundational architecture for artificial general intelligence, *Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms*, **6**, p. 36.

FREESCALE. (2001). C-5 Network Processor Architecture Guide.

FROST-J., NUMAN-M. W., LIEBELT-M., AND PHILLIPS-B. J. (2015). A new computer for cognitive computing, *14th IEEE International Conference on Cognitive Informatics & Cognitive Computing*, Beijing, China.

GAMBLE-R. F. (1990). Transforming rule-based programs: from the sequential to the parallel, *Proceedings of the 3rd international conference on Industrial and engineering applications of artificial intelligence and expert systems-Volume 2*, ACM, pp. 854–863.

GARBERS-J., PROMEL-H. J., AND STEGER-A. (1990). Finding clusters in VLSI circuits, *1990 IEEE International Conference on Computer-Aided Design*, pp. 520–523.

GAREY-M. R., AND JOHNSON-D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA.

GENDRON-B., AND CRAINIC-T. G. (1994). Parallel branch-and-branch algorithms: Survey and synthesis, *Oper. Res.*, **42**(6), pp. 1042–1066.

GILBERT-J. R., AND ZMIJEWSKI-E. (1987). A parallel graph partitioning algorithm for a message-passing multiprocessor, *Int. J. Parallel Program.*, **16**(6), pp. 427–449.

GLASS-C. J., AND NI-L. M. (1992). The turn model for adaptive routing, *SIGARCH Comput. Archit. News*, **20**(2), pp. 278–287.

GOERTZEL-B., AND WANG-P. (2007). *Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms: Proceedings of the AGI Workshop 2006*, Vol. 157, IOS Press.

GOLDBERG-D. E. (1994). Genetic and evolutionary algorithms come of age, *Communications of the ACM*, **37**(3), pp. 113–120.

GUERRIER-P., AND GREINER-A. (2000). A generic architecture for on-chip packet-switched interconnections, *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pp. 250–256.

GUPTA-A. (1985). *Parallelism in Production Systems*, PhD thesis, Carnegie-Mellon University.

HAGEN-L., AND KAHNG-A. (1991). Fast spectral methods for ratio cut partitioning and clustering, *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference on*, pp. 10–13.

HARMANANI-H. M., AND FARAH-R. (2008). A method for efficient mapping and reliable routing for NoC architectures with minimum bandwidth and area, *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, IEEE, pp. 29–32.

HAWKINS-J., AND BLAKESLEE-S. (2004). *On intelligence: How a New Understanding of the Brain will Lead to the Creation of Truly Intelligent Machines*, Times Books.

HAWKINS-J., AND GEORGE-D. (2006). Hierarchical temporal memory: Concepts, theory and terminology, *Technical report*, Technical report, Numenta.

HAYES-ROTH-F., AND JACOBSTEIN-N. (1994). The state of knowledge-based systems, *Communications of the ACM*, **37**(3), pp. 26–39.

HAYKIN-S. S. (2001). *Neural networks: a comprehensive foundation*, Tsinghua University Press.

HEATH-M. T., AND RAGHAVAN-P. (1995). A cartesian parallel nested dissection algorithm, *SIAM Journal on Matrix Analysis and Applications*, **16**(1), pp. 235–253.

HECHT-NIELSEN-R. (2007). *Confabulation Theory: The Mechanism of Thought*, Springer.

HENDRICKSON-B., AND LELAND-R. (1995). A multilevel algorithm for partitioning graphs, *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, ACM, New York, NY, USA.

HENNESSY-J. L., AND PATTERSON-D. A. (2011). *Computer architecture: a quantitative approach*, Elsevier.

HOYA-T. (2005). *Artificial Mind System. Kernel Memory Approach*, Springer.

HU-J., AND MARCULESCU-R. (2003a). Energy-aware Mapping for Tile-based NoC Architectures Under Performance Constraints, *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, ACM, New York, NY, USA, pp. 233–239.

HU-J., AND MARCULESCU-R. (2003b). Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures, *Design, Automation and Test in Europe Conference and Exhibition, 2003*, IEEE, pp. 688–693.

HU-J., AND MARCULESCU-R. (2004). DyAD - smart routing for networks-on-chip, *Design Automation Conference, 2004. Proceedings. 41st*, pp. 260–263.

HU-J., AND MARCULESCU-R. (2005). Energy- and performance-aware mapping for regular NoC architectures, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, **24**(4), pp. 551–562.

HUTTER.-M. (2004). *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*, Springer.

IBM. (2005). An architectural blueprint for autonomic computing, *Technical report*, IBM.

ISHIDA-T. (1990). Methods and effectiveness of parallel rule firing, *Artificial Intelligence Applications, 1990., Sixth Conference on*, IEEE, pp. 116–122.

ISHIWATA-S., YAMAKAGE-T., TSUBOI-Y., SHIMAZAWA-T., KITAZAWA-T., MICHINAKA-S., YAHAGI-K., TAKEDA-H., OUE-A., KODAMA-T., MATSUMOTO-N., KAMEI-T., SAITO-M., MIYAMORI-T., OOTOMO-G., AND MATSUI-M. (2003). A single-chip MPEG-2 codec based on customizable media embedded processor, *IEEE Journal of Solid-State Circuits*, **38**(3), pp. 530–540.

ITRS. (2015). International technology roadmap for semiconductors, http://www.itrs2.net/itrs-reports.html.

JACKSON-P. (1986). *Introduction to expert systems*, Addison-Wesley Pub. Co.,Reading, MA.

JALABERT-A., MURALI-S., BENINI-L., AND DE MICHELI-G. (2004). XpipesCompiler: a tool for instantiating application specific networks on chip, *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, Vol. 2, IEEE, pp. 884–889.

JANG-W., AND PAN-D. Z. (2010). A3MAP: Architecture-Aware Analytic Mapping for Networks-on-Chip, *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 523–528.

JENA-R. K., AND SHARMA-G. K. (2007). A multiobjective evolutionary algorithm-based optimisation model for network on chip synthesis, *International Journal of Innovative Computing and Applications*, **1**(2), pp. 121–127.

JERRAYA-A., AND WOLF-W. (2004). *Multiprocessor systems-on-chips*, Elsevier.

JETLY-K. (2013). Experimental comparison of store-and-forward and wormhole noc routers for fpga's.

JONES-G. (1998). Genetic and evolutionary algorithms, *Encyclopedia of Computational Chemistry*, **2**, pp. 1127–1136.

KARYPIS-G., AND KUMAR-V. (1995). Multilevel graph partitioning schemes, *Proc. 24th Intern. Conf. Par. Proc., III*, CRC Press, pp. 113–122.

KARYPIS-G., AND KUMAR-V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM J. Sci. Comput.*, **20**(1), pp. 359–392.

KERMANI-P., AND KLEINROCK-L. (1979). Virtual cut-through: A new computer communication switching technique, *Computer Networks (1976)*, **3**(4), pp. 267–286.

KERNIGHAN-B. W., AND LIN-S. (1970). An efficient heuristic procedure for partitioning graphs, *The Bell System Technical Journal*, **49**(2), pp. 291–307.

KIERAS-D. E., AND MEYER-D. E. (1997). An overview of the epic architecture for cognition and performance with application to human-computer interaction, *Hum.-Comput. Interact.*, **12**(4), pp. 391–438.

KIRKPATRICK-S., GELATT-C. D., AND VECCHI-M. P. (1983). Optimization by simulated annealing, *Science*, **220**(4598), pp. 671–680.

KOEDINGER-K. R., ANDERSON-J. R., HADLEY-W. H., AND MARK-M. A. (1997). Intelligent Tutoring Goes to School in the Big City, *International Journal of Artificial Intelligence in Education*, pp. 30–43.

KOKINOV-B. N. (1994). The DUAL Cognitive Architecture: A Hybrid Multi-Agent Approach, *ECAI*, pp. 203–207.

KOZIRIS-N., ROMESIS-M., TSANAKAS-P., AND PAPAKONSTANTINOU-G. (2000). An efficient algorithm for the physical mapping of clustered task graphs onto multiprocessor architectures, *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*, IEEE, pp. 406–413.

KUMAR-S. (2013). Introducing Qualcomm Zeroth Processors: Brain-Inspired Computing, https://www.qualcomm.com/news/onq/2013/10/10/introducing-qualcomm-zeroth-processors-brain-inspired-computing.

KUMAR-S., JANTSCH-A., SOININEN-J. P., FORSELL-M., MILLBERG-M., OBERG-J., TIENSYRJA-K., AND HEMANI-A. (2002). A network on chip architecture and design methodology, *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pp. 105–112.

KUO-S., AND MOLDOVAN-D. (1992). The state of the art in parallel production systems, *Journal of Parallel and Distributed Computing*, **15**(1), pp. 1–26.

LAHIRI-K., RAGHUNATHAN-A., AND DEY-S. (2001). Evaluation of the traffic-performance characteristics of system-on-chip communication architectures, *VLSI Design, 2001. Fourteenth International Conference on*, IEEE, pp. 29–35.

LAIRD-J. E. (2012). *The Soar Cognitive Architectures*, The MIT Press, Cambridge.

LAIRD-J. E., NEWELL-A., AND ROSENBLOOM-P. S. (1987). Soar: An architecture for general intelligence, *Artificial Intelligence*, **33**, pp. 1–64.

LAIRD-J. E., ROSENBLOOM-P. S., AND NEWELL-A. (1986). Chunking in Soar: The anatomy of a general learning mechanism, *Machine Learning*, **1**, pp. 11–46.

LANDMANN-N., KUHN-M., PIOSCZYK-H., FEIGE-B., BAGLIONI-C., SPIEGELHALDER-K., FRASE-L., RIEMANN-D., STERR-A., AND NISSEN-C. (2014). The reorganisation of memory during sleep, *Sleep Medicine Reviews*, **18**(6), pp. 531–541.

LANGLEY-P. (2006). Cognitive architectures and general intelligent systems, *AI magazine*, **27**(2), p. 33.

LANGLEY-P., CHOI-D., AND ROGERS-S. (2009a). Acquisition of hierarchical reactive skills in a unified cognitive architecture, *Cognitive Systems Research*, **10**(4), pp. 316 – 332.

LANGLEY-P., LAIRD-J. E., AND ROGERS-S. (2009b). Cognitive Architectures: Research Issues and Challenges, *Cognitive Systems Research*, **10**, pp. 141–160.

LAWLER-E. L., AND WOOD-D. E. (1966). Branch-and-bound methods: A survey, *Operations research*, **14**(4), pp. 699–719.

LEI-T., AND KUMAR-S. (2003). A two-step genetic algorithm for mapping task graphs to a network on chip architecture, *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, IEEE, pp. 180–187.

LEWIS-R. L. (1993). An architecturally-based theory of human sentence comprehension, *Proceedings of the fifteenth annual conference of the cognitive science society*, pp. 108–113.

LINDER-D. H., AND HARDEN-J. C. (1991). An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes, *IEEE Transactions on Computers*, **40**(1), pp. 2–12.

LINES-A. (2004). Asynchronous interconnect for synchronous SoC design, *IEEE Micro*, **24**(1), pp. 32–41.

LIN-T., AND PILEGGI-L. T. (2002). Throughput-driven IC communication fabric synthesis, *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pp. 274–279.

LIN-T.-J., LIN-S.-Y., AND WU-A.-Y. (2008). Traffic-balanced IP mapping algorithm for 2D-Mesh on-chip-networks, *2008 IEEE Workshop on Signal Processing Systems*, IEEE, pp. 200–203.

LUGER-G. (2004). *Artificial Intelligence: Structures and Strategies for Complex Problem Solving (5th Edition)*, Pearson Addison Wesley.

LU-Z., XIA-L., AND JANTSCH-A. (2008). Cluster-based Simulated Annealing for Mapping Cores onto 2D Mesh Networks on Chip, *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on*, pp. 1–6.

MAGERKO-B., LAIRD-J. E., ASSANIE-M., KERFOOT-A., AND STOKES-D. (2004). AI Characters and Directors for Interactive Computer Games, *Proceedings of the 16th Conference on Innovative Applications of Artifical Intelligence*, IAAI'04, AAAI Press, pp. 877–883.

MANG-E., MANG-I., AND CONSTANTIN-P. R. (2009). Reconfigurable computing – a new paradigm, *Journal of Computer Science and Control Systems*, **2**, pp. 22–27.

MARCULESCU-R., OGRAS-U. Y., PEH-L. S., JERGER-N. E., AND HOSKOTE-Y. (2009). Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **28**(1), pp. 3–21.

MARTIN-G., BAILEY-B., AND PIZIALI-A. (2010). *ESL design and verification: a prescription for electronic system level methodology*, Morgan Kaufmann.

MCDERMOTT-J. P. (1980). RI: an Expert in the Computer Systems Domain, *AAAI*, Vol. 1, pp. 269–271.

MEROLLA-P. A., ARTHUR-J. V., ALVAREZ-ICAZA-R., CASSIDY-A. S., SAWADA-J., AKOPYAN-F., JACKSON-B. L., IMAM-N., GUO-C., NAKAMURA-Y., BREZZO-B., VO-I., ESSER-S. K., APPUSWAMY-R., TABA-B., AMIR-A., FLICKNER-M. D., RISK-W. P., MANOHAR-R., AND MODHA-D. S. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface, *Science*, **345**(6197), pp. 668–673.

MILLBERG-M., NILSSON-E., THID-R., KUMAR-S., AND JANTSCH-A. (2004). The nostrum backbone-a communication protocol stack for networks on chip, *VLSI Design, 2004. Proceedings. 17th International Conference on*, IEEE, pp. 693–696.

MILLER-C. S., AND LAIRD-J. E. (1996). Accounting for graded performance within a discrete search framework, *Cognitive Science*, **20**(4), pp. 499 – 537.

MILLER-G. L., TENG-S.-H., AND VAVASIS-S. A. (1991). A unified geometric approach to graph separators, *Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science*, SFCS '91, IEEE Computer Society, Washington, DC, USA, pp. 538–547.

MINSKY-M. (1974). A framework for representing knowledge, *Technical report*, Cambridge, MA, USA.

MIRANKER-D., AND LOFASO-B. (1991). The organization and performance of a TREAT-based production system compiler, *Knowledge and Data Engineering, IEEE Transactions on*, **3**(1), pp. 3–10.

MIRANKER-D. P. (1987). TREAT: A better match algorithm for AI production systems, *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, AAAI Press, pp. 42–47.

MIRANKER-D. P., BRANT-D., LOFASO-B. J., AND GADBOIS-D. (1990). On the performance of lazy matching in production systems, *Proc. 1990 Nat. Conf. Artif. Intell.*, pp. 685 –692.

MODARRESSI-M., AND SARBAZI-AZAD-H. (2012). Reconfigurable Cluster-Based Networks-on-Chip for Application-Specific MPSoCs, *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pp. 153–156.

MOORE-G. E. (1965). Cramming more components onto integrated circuits, *Electronics Magazine*, pp. 114–117.

MOORE-G. E. (1998). Cramming more components onto integrated circuits, *Proceedings of the IEEE*, **86**, pp. 82–85.

MORAES-F., CALAZANS-N., MELLO-A., MLLER-L., AND OST-L. (2004). HERMES: an infrastructure for low area overhead packet-switching networks on chip, *Integration, the VLSI Journal*, **38**(1), pp. 69 – 93.

MURALI-S., AND DE MICHELI-G. (2004). SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs, *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, ACM, New York, NY, USA, pp. 914–919.

MURALI-S., AND MICHELI-G. D. (2004). Bandwidth-constrained mapping of cores onto NoC architectures, , **2**, pp. 896–901 Vol.2.

NAYAK-P., GUPTA-A., AND ROSENBLOOM.-P. (1988). Comparison of the Rete and Treat production matchers for soar - a summary, *Proceedings of the National Conference on Artificial Intelligence*, pp. 693–698.

NEWELL-A. (1990). *Unified theories of cognition*, Harvard University, Cambridge, MA.

NEWELL-A., AND SIMON-A. H. (1972). *Human problem solving*, Prentice-Hall.

NEWELL-A., AND SIMON-H. A. (1976). Computer science as empirical inquiry: Symbols and search, *Commun. ACM*, **19**(3), pp. 113–126.

NILSSON-N. J. (2014). *Principles of artificial intelligence*, Morgan Kaufmann.

NUMAN-M. W., FROST-J., PHILLIPS-B. J., AND LIEBELT-M. (2015). A network-based communication platform for a cognitive computer, *International Workshop on Artificial Intelligence and Cognition (AIC 2015)*, Turin, Italy.

OFLAZER-K. (1986). *Partitioning and parallel processing of production systems*, PhD thesis, Carnegie-Mellon University.

ORACLE. (2012). SPARC T3 Data Sheet, http://www.oracle.com/products.

O'REILLY-R. C., AND MUNAKATA-Y. (2000). *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain*, 1st edn, MIT Press, Cambridge, MA, USA.

PASIK-A. J. (1989). *A methodology for programming production systems and its implications on parallelism*, PhD thesis.

PEI-W. (2007). Artificial general intelligence: A gentle introduction, *Mind and Computation,(2)*, pp. 213–222.

PHILLIPS-B. J. (2016). The Street Project, https://bitbucket.org/bjphillips/javastreet.

PICOCHIP. (2012). PC202 processor.

POP-R., AND KUMAR-S. (2004). A survey of techniques for mapping and scheduling applications to network on chip systems, *School of Engineering, Jonkoping University, Research Report*, **4**, p. 4.

POTHEN-A., SIMON-H. D., AND LIOU-K.-P. (1990). Partitioning sparse matrices with eigenvectors of graphs, *SIAM J. Matrix Anal. Appl.*, **11**(3), pp. 430–452.

RADU-C., AND VINŢAN-L. (2013). *Domain-Knowledge Optimized Simulated Annealing for Network-on-Chip Application Mapping*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 473–487.

RADU-C., MAHBUB-M. S., AND VINTAN-L. (2013). Developing domain-knowledge evolutionary algorithms for network-on-chip application mapping, *Microprocessors and Microsystems*, **37**(1), pp. 65 – 78.

RASCHID-L., SELLIS-T., AND LIN-C. C. (1988). Exploiting concurrency in a DBMS implementation for production systems, *Proc. Int. Symp. Databases Parallel Distributed Syst.*

RESHADI-M., KHADEMZADEH-A., AND REZA-A. (2010). Elixir: a new bandwidth-constrained mapping for networks-on-chip, *IEICE Electronics Express*, **7**(2), pp. 73–79.

RICHARDSON-T., NICOPOULOS-C., PARK-D., NARAYANAN-V., XIE-Y., DAS-C., AND DEGALAHAL-V. (2006). A hybrid SoC interconnect with dynamic TDMA-based transaction-less buses and on-chip networks, *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on.*

RICH-E., AND KNIGHT-K. (1991). *Artificial intelligence*, Vol. 199, McGraw-Hill New York:.

RIJPKEMA-E., GOOSSENS-K. G. W., RADULESCU-A., DIELISSEN-J., VAN MEERBERGEN-J., WIELAGE-P., AND WATERLANDER-E. (2003a). Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip, , pp. 350–355.

RIJPKEMA-E., GOOSSENS-K., RADULESCU-A., DIELISSEN-J., VAN MEERBERGEN-J., WIELAGE-P., AND WATERLANDER-E. (2003b). Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip, *IEE Proceedings-Computers and Digital Techniques*, **150**(5), pp. 294–302.

SAEIDI-S., KHADEMZADEH-A., AND MEHRAN-A. (2007). SMAP: An intelligent mapping tool for network on chip, *2007 International Symposium on Signals, Circuits and Systems*, Vol. 1, IEEE, pp. 1–4.

SAHU-P. K., AND CHATTOPADHYAY-S. (2013). A survey on application mapping strategies for network-on-chip design, *Journal of Systems Architecture*, **59**(1), pp. 60–76.

SAHU-P. K., SHAH-N., MANNA-K., AND CHATTOPADHYAY-S. (2010). A new application mapping algorithm for mesh based network-on-chip design, *2010 Annual IEEE India Conference (INDICON)*, pp. 1–4.

SAHU-P. K., SHAH-T., MANNA-K., AND CHATTOPADHYAY-S. (2014). Application mapping onto mesh-based network-on-chip using discrete particle swarm optimization, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **22**(2), pp. 300–312.

SAPONARA-S., AND FANUCCI-L. (2012). Homogeneous and heterogeneous MPSoC architectures with Network-On-Chip connectivity for low-power and real-time multimedia signal processing, *VLSI Design*, **2012**, p. 16.

SCHERER-T. (2015). 50 Years of Moores Law, https://www.elektormagazine.com/articles/moores-law.

SCHOR-M. I., DALY-T. P., LEE-H. S., AND TIBBITTS-B. (1986). Advances in Rete pattern matching, *AAAI*, Vol. 86, pp. 226–232.

SGROI-M., SHEETS-M., MIHAL-A., KEUTZER-K., MALIK-S., RABAEY-J., AND SANGIOVANNI-VENCENTELLI-A. (2001). Addressing the system-on-a-chip interconnect woes through communication-based design, *Proceedings of the 38th annual Design Automation Conference*, ACM, pp. 667–672.

SHANG-L., PEH-L.-S., AND JHA-N. K. (2006). PowerHerd: a distributed scheme for dynamically satisfying peak-power constraints in interconnection networks, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **25**(1), pp. 92–110.

SHAPIRO-S. C., RAPAPORT-W. J., KANDEFER-M., JOHNSON-F. L., AND GOLDFAIN-A. (2007). Metacognition in SNePS, *AI Magazine*, **28**(1), p. 17.

SHORTLIFFE-E. (2012). *Computer-based medical consultations: MYCIN*, Vol. 2, Elsevier.

SHU-C. (2014). Google Acquires Artificial Intelligence Startup DeepMind For More Than $500M, http://techcrunch.com/2014/01/26/google-deepmind.

SIEBENBORN-A., BRINGMANN-O., AND ROSENSTIEL-W. (2004). Communication analysis for network-on-chip design, *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, PARELEC '04, IEEE Computer Society, Washington, DC, USA, pp. 315–320.

SIMON-H. (1991). Parallel methods on large-scale structural analysis and physics applications partitioning of unstructured problems for parallel processing, *Computing Systems in Engineering*, **2**(2), pp. 135 – 148.

SIMON-H. A., AND NEWELL-A. (1958). Heuristic problem solving: The next advance in operations research, *Operations Research*, **6**(1), pp. 1–10.

SOWA-J. F. (1992). Conceptual graphs as a universal knowledge representation, *Computers & Mathematics with Applications*, **23**(2), pp. 75–93.

SRINIVASA-N., AND CRUZ-ALBRECHT-J. M. (2012). Neuromorphic adaptive plastic scalable electronics: analog learning systems, *IEEE pulse*, **3**(1), pp. 51–56.

STOLFO-S. (1983). The DADO parallel computer, *Technical report*, Technical report, Department of Computer Science, Columbia University, New York.

STOLFO-S. (1987). Initial performance of the DADO2 prototype, *Computer; (United States)*.

STOLFO-S. J., DEWAN-H. M., AND WOLFSON-O. (1991). The PARULEL Parallel Rule Language, *ICPP (2)*, pp. 36–45.

SUN-R. (2001). *Duality of the mind: A bottom-up approach toward cognition*, Psychology Press.

SUN-R. (2006). The CLARION cognitive architecture: Extending cognitive modeling to social simulation, *Cognition and multi-agent interaction*, pp. 79–99.

SUN-R., AND ALEXANDRE-F. (2013). *Connectionist-symbolic integration: From unified to hybrid approaches*, Psychology Press.

TAATGEN-N. A. (2005). Modeling parallelization and speed improvement in skill acquisition: From dual tasks to complex dynamic skills, *Cognitive Science*, **29**, pp. 421–455.

TAMBE-M., JOHNSON-W. L., JONES-R. M., KOSS-F., LAIRD-J. E., AND ROSENBLOOM-P. S. (1995). Intelligent agents for interactive simulation environments, *AI Magazine*, **16**, pp. 15–39.

TEXAS INSTRUMENTS INC.. (2004). OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide.

THORISSON-K. R., AND HELGASSON-H. P. (2012). Cognitive architectures and autonomy: A comparative review, *Journal of Artificial General Intelligence*, **3**(2), pp. 1–30.

TOSUN-S. (2011). Cluster-based application mapping method for network-on-chip, *Advances in Engineering Software*, **42**(10), pp. 868 – 874.

TRAFTON-J. G., CASSIMATIS-N. L., BUGAJSKA-M. D., BROCK-D. P., MINTZ-F. E., AND SCHULTZ-A. C. (2005). Enabling effective human-robot interaction using perspective-taking in robots, *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, **35**(4), pp. 460–470.

TURING-A. M. (1950). Computing machinery and intelligence, *Mind*, **59**(236), pp. 433–460.

WALTER-I., CIDON-I., KOLODNY-A., AND SIGALOV-D. (2009). The Era of Many-modules SoC: Revisiting the NoC Mapping Problem, *Proceedings of the 2Nd International Workshop on Network on Chip Architectures*, NoCArc '09, ACM, New York, NY, USA, pp. 43–48.

WANG-J., LI-Y., CHAI-S., AND PENG-Q. (2011). Bandwidth-aware application mapping for NoC-based MPSoCs, *Journal of Computational Information Systems*, **7**(1), pp. 152–159.

WANG-P. (1995). *Non-Axiomatic Reasoning System: Exploring the Essence of Intelligence*, PhD thesis, Dept. of Computer Science, Indiana Univ., CITY, Indiana.

WANG-P. (2006). *Rigid flexibility: The Logic of Intelligence*, Springer.

WATERMAN-D. (1986). *A guide to expert systems*, Addison-Wesley.

WENBIAO-Z., ZHANG-Y., SHENZHEN-G., MAO-Z., AND HARBIN-H. (2007). Link-load balance aware mapping and routing for NoC, *Architecture*, **4**(5), p. 6.

WESTE-N., AND HARRIS-D. (2010). *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th edn, Addison-Wesley Publishing Company, USA.

WIKLUND-D., AND LIU-D. (2003). SoCBUS: switched network on chip for hard real time embedded systems, *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, IEEE, pp. 8–pp.

WINGARD-D. (2001). MicroNetwork-based integration for SOCs, *Design Automation Conference, 2001. Proceedings*, IEEE, pp. 673–677.

WOLKOTTE-P. T., SMIT-G. J. M., RAUWERDA-G. K., AND SMIT-L. T. (2005). An energy-efficient reconfigurable circuit-switched network-on-chip, *19th IEEE International Parallel and Distributed Processing Symposium*, pp. 155a–155a.

YAMAUCHI-H., OKADA-S., TAKETA-K., MATSUDA-Y., MORI-T., OKADA-S., WATANABE-T., HARADA-Y., MATSUDAIRA-M., AND MATSUSHITA-Y. (2002). A 0.8 W HDTV video processor with simultaneous decoding of two MPEG2 MP@HL streams and capable of 30 frames/s reverse playback, *2002 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No.02CH37315)*, Vol. 1, pp. 372–474 vol.1.

YE-T. T., BENINI-L., AND MICHELI-G. D. (2002). Analysis of power consumption on switch fabrics in network routers, *Design Automation Conference, 2002. Proceedings. 39th*, pp. 524–529.

ZACKS. (2013). Yahoo acquired SkyPhrase, http://finance.yahoo.com/news/yahoo-acquires-skyphrase-204002614.html.

ZADEH-L. A. (1994). Fuzzy logic, neural networks, and soft computing, *Communications of the ACM*, **37**(3), pp. 77–85.

ZIMMERMANN-H. (1980). OSI reference model-the ISO model of architecture for open systems interconnection, *IEEE Transactions on Communications*, **28**(4), pp. 425–432.

List of Acronyms

ACT-R Adaptive Control of Thought-Rational

AG Architecture graph

AGI Artificial general intelligence

AI Artificial intelligence

BB Branch and Bound

CAM Content addressable memory

CE Condition element

FSM Finite state machine

GPE Group of processing elements

IC Integrated circuit

LBC Lower bound cost

MPSoC Multi-processor system on chip

NoC Network on chip

OSI Open systems interconnection

PE Processing element

PSA Priority-based Simulated Annealing

PTG Partitioned traffic graph

RCAM Relational content addressable memory

SA Simulated Annealing

Soar State, Operator And Result

SoC System on chip

TG	Traffic graph
UBC	Upper bound cost
WME	Working memory element

Publications

Conference Articles

NUMAN-M. W., FROST-J., PHILLIPS-B. J., AND LIEBELT-M. (2015). A Network-based Communication Platform for a Cognitive Computer, *International Workshop on Artificial Intelligence and Cognition (AIC 2015)*, Turin, Italy.

FROST-J., **NUMAN-M. W.**, LIEBELT-M., AND PHILLIPS-B. J. (2015). A New Computer for Cognitive Computing, *14th IEEE International Conference on Cognitive Informatics & Cognitive Computing*, Beijing, China.

Journal Articles in Preparation

NUMAN-M. W., PHILLIPS-B. J., AND LIEBELT-M. Priority-based Network on Chip Mapping of the Processing Elements of Production Systems.

FROST-J., WANG-P., **NUMAN-M. W.**, LIEBELT-M., AND PHILLIPS-B. Relational Content Addressable Memory.

Biography

Mostafa Wasiuddin Numan was born in Bangladesh in 1983. He holds a B.Sc. (Engineering) degree in Computer Science and Engineering from Shahjalal University of Science and Technology, Bangladesh (2006) and an M.Sc. (By research) degree in Electrical, Electronic and Systems Engineering from National University of Malaysia (UKM), Malaysia (2010). He worked towards his Ph.D. degree under the supervision of Professor Michael Liebelt and Dr Braden J. Phillips at the School of Electrical and Electronic Engineering of the University of Adelaide (2013 – 2017). His research focused on a computer architecture development for artificial intelligence (AI) and cognitive systems. He worked on Network on Chip (NoC) based communication platform of the system. In his research, he proposed a novel optimisation algorithm to map the processing elements onto the NoC platform. His research findings were published in reputed peer-reviewed journals and international conferences. During Ph.D. candidature, Mr Numan received the International Postgraduate Research Scholarship (IPRS) and Australian Postgraduate Award (APA) from the Australian Government. Previously, he was awarded the prestigious Yayasan Khazanah Scholarship by the Malaysian Government during M.Sc. study. He was awarded the Travel Grant Award by IEEE International Symposium on Communication System, Networks and Digital Signal Processing, UK (2010). In addition to research, Mr Numan also possess experiences of working in software and telecommunication industry in Bangladesh, Malaysia and Australia.

Mostafa Wasiuddin Numan
mostafa.numan@adelaide.edu.au