ΕΘΝΙΚΟ ΜΕΤΣΟΒΕΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

DESIGN AND IMPLEMENTATION OF A RUN-TIME MANAGER FOR NETWORK-ON-CHIP (NoC) ARCHITECTURES

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Χ. Καθάρειος

Επιβλέπων: Δημήτριος Σούντρης

Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2011

ΕΘΝΙΚΟ ΜΕΤΣΟΒΕΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

DESIGN AND IMPLEMENTATION OF A RUN-TIME MANAGER FOR NETWORK-ON-CHIP (NoC) ARCHITECTURES

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Χ. Καθάρειος

Επιβλέπων: Δημήτριος Σούντρης

Επ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21^{η} Οκτωβρίου 2011

....	....	....
Δημήτριος Σούντρης	Κιαμάλ Ζ. Πεκμεστζή	Γεώργιος Οικονομάκος
Επ. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2011

....
Γεώργιος Χ. Καθάρειος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικών Υπολογιστών Ε.Μ.Π.

Copyright © Γεώργιος Χ. Καθάρειος, 2011.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Αντικείμενο της παρούσας διπλωματικής εργασίας αποτελεί η μελέτη και η ανάπτυξη ενός διαχειριστή πόρων ενός Πολυ-Πύρηνου Συστήματος-σε-Ψηφίδα (Multi-Processor-System-on-Chip, MPSoC) που χρησιμοποιεί δίκτυο διασύνδεσης (interconnection network, ICN) τύπου αρχιτεκτονικής Δικτύου-σε-Ψηφίδα (Network-on-Chip, NoC). Η εργασία επικεντρώνεται στην ανάπτυξη ενός αλγορίθμου που έχει σκοπό τον υπολογισμό της κατά το δυνατότερο αποδοτικότερης χαρτογράφησης στον χρόνο εκτέλεσης (run-time mapping) των διεργασιών μίας εφαρμογής που πρόκειται να εκτελεστεί στο εν λόγω σύστημα, προκειμένου να ελαχιστοποιείται η κατανάλωση ενέργειας και να μεγιστοποιείται η απόδοση του συστήματος.

Στο κεφάλαιο 1, παρουσιάζονται τα βασικά χαρακτηριστικά και ο τρόπος λειτουργίας ενός Δικτύου-σε-Ψηφίδα. Παρουσιάζεται η έννοια της χαρτογράφησης μια εφαρμογής και αναλύονται έννοιες που θα χρησιμοποιούνται στην συνέχεια, όπως τα ομογενή και ετερογενή Δίκτυα-σε-Ψηφίδα, ο γράφος των διεργασιών μίας εφαρμογής κλπ.

Στο κεφάλαιο 2, παρουσιάζονται οι κορυφαίες στον χώρο τους σχετικές εργασίες που ασχολούνται με την χαρτογράφηση στον χρόνο εκτέλεσης. Δίνεται έμφαση στην καινοτόμο ιδέα της καθεμίας και στο τέλος του κεφαλαίου γίνεται μία συνοπτική σύγκριση μεταξύ τους.

Στο κεφάλαιο 3, περιγράφεται ο αλγόριθμος υπολογισμού της χαρτογράφησης στον χρόνο εκτέλεσης που αναπτύχθηκε και υλοποιήθηκε στα πλαίσια της διπλωματικής εργασίας. Αναλύεται σε δύο σκέλη, το πρώτο αναφέρεται σε ομογενή και το δεύτερο σε ετερογενή συστήματα.

Στο κεφάλαιο 4, γίνεται αρχικά παρουσίαση της πλατφόρμας που χρησιμοποιήθηκε για την εξαγωγή αποτελεσμάτων και στην συνέχεια συγκρίνεται ο υλοποιημένος αλγόριθμος με άλλους state-of-the-art αλγορίθμους.

Στο κεφάλαιο 5 ανακεφαλαιώνονται τα συμπεράσματα της διπλωματικής, και παρουσιάζονται κάποια θέματα και ιδέες για διερεύνηση και μελλοντική έρευνα.

Λέξεις Κλειδιά

Σύστημα-σε-Ψηφίδα, Πολυ-Πύρηνο Σύστημα-σε-Ψηφίδα, Δίκτυο-σε-Ψηφίδα, Χαρτογράφηση στον χρόνο εκτέλεσης, Ελαχιστοποίηση κατανάλωσης ισχύος

Abstract

The purpose of this diploma thesis is the design and implementation of a run-time resource manager for a Multi-Processor System-on-Chip (MPSoC) that utilizes the Network-on-Chip (NoC) architecture. The thesis focuses on the implementation of an algorithm that aims at computing the best possible mapping on run-time, for the tasks of an application that is going to be executed on the system, in order to minimize the energy consumption, while maximizing the performance of the system.

In chapter 1, we make an introduction on the basic characteristics and functions of a Network-on-Chip. We present the concept of application mapping and analyze terms that will be needed in the following, such as homogeneous and heterogeneous Networks-on-Chips, the Application Task Graph etc.

In chapter 2, four published works of state-of-the-art run-time mapping algorithms are presented. Emphasis is given on the innovative contribution of each paper and a comparison between them concludes the chapter.

In chapter 3, the run-time mapping algorithm that was developed as part of this thesis is described. It is analyzed in two parts, each of which deals with homogeneous and heterogeneous systems respectively.

In chapter 4, initially the platform used for the experimental results is presented and following is the comparison of our Run-Time Mapping algorithm with other state-of-the-art algorithms.

Finally in chapter 5, we summarize the conclusions of the diploma thesis and present some topics and ideas for future work and research.

Keywords

System-on-Chip, Multi-Processor System-on-Chip, Network-on-Chip, run-time mapping, Energy consumption minimization

Ευχαριστίες/Acknowledgements

Για την εκπόνηση της παρούσας διπλωματικής εργασίας θα ήθελα να εκφράσω τις ειλικρινείς μου ευχαριστίες προς τον επιβλέποντα καθηγητή κ. Δ. Σούντρη ο οποίος εμπιστεύθηκε στο πρόσωπο μου την ανάθεση ενός ιδιαίτερα ενδιαφέροντος και απαιτητικού επιστημονικού έργου.
Επιπλέον, αυτή η εργασία δεν θα είχε έλθει εις πέρας χωρίς την πολύτιμη βοήθεια και καθοδήγηση του υποψήφιου διδάκτορα Ηρακλή Αναγνωστόπουλου και του διδάκτορα Αλέξανδρου Μπάρτζα που βοήθησαν τα μέγιστα με τις συμβουλές, τις γνώσεις και την υπομονή τους.

Table of Contents

Chapter 1: Networks-on-Chip **11**

1.1. Introduction 13

1.2. Network-on-Chip 14

1.2.1. Homogeneity and Granularity 16

1.3. Network layers 18

1.3.1. System Layer 19

1.3.2. Network Interface Layer 20

1.3.3. Network Layer 20

1.3.4. Link Layer 23

1.4. Run-Time Mapping 25

1.4.1. The Cost Function 25

1.4.2. Application Task Graph 26

1.4.3. Run-Time and Design-Time Mapping 27

1.4.4. Distributed and Centralized Mapping 27

Chapter 2: State-of-the-art Run-time mapping algorithms **31**

2.1. ADAM: Run-time Agent-based Distributed Application Mapping for on-chip Communication 33

2.2. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles 35

2.3. Incremental Run-time Application Mapping for Homogeneous NoCs with Multiple Voltage Levels 37

2.4. Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip 39

2.5. Comparison Table 41

Chapter 3: Run-Time Mapping (RTM) Algorithms **43**

3.1. Main Idea behind the Run-Time Mapping (RTM) Algorithms 45

3.2. Run-Time Mapping (RTM) on a homogeneous platform 48
3.2.1. Definitions 48
3.2.2. The Run-time mapping algorithm for homogeneous NoCs 49
3.2.3. Example of the execution of the RTM algorithm on a homogeneous platform 51
3.3. Run-Time Mapping (RTM) on a heterogeneous platform 52
3.3.1. Additional definitions 52
3.3.2. The Algorithm for heterogeneous NoCs 52
3.3.3. Example of the execution of the RTM algorithm on a heterogeneous platform 56

Chapter 4: Experimental Results **59**
4.1. Introduction 61
4.2. The Application Platform 61
4.3. Experimental Results of the RTM algorithm for homogeneous platforms 65
4.3.1. TGFF generated applications 65
4.3.2. Application Benchmarks 67
4.4. Experimental Results of the RTM algorithm for heterogeneous platforms 69
4.4.1. TGFF generated applications 69
4.4.2. Utilization Scenarios 71
4.5. Experimental results' conclusions 72

Chapter 5: Conclusions and Future work **75**
5.1. Summary 77
5.2. Future Work 77
5.2.1. Task Migration 77
5.2.2. Multitasking on the cores, Spatial and Temporal mapping 78
5.2.3. High-level NoC control mechanisms for run time mapping 79

References **81**

Chapter 1:
Networks-on-Chip

1.1. Introduction

Recent advances in VLSI technology have made the transition from single-core architectures to multi-core ones imperative. The level of integration allows us to have several Processing Element (PE) units in one chip and thus manufacturers tend to integrate more elements, in order to achieve highest performance and to satisfy the more and more demanding applications in the market. According to Moore's law (fig. 1.1), it is not unlikely to see thousands of processors in a single chip in the recent future. That being said, it is evident that the communication between these processors cannot be efficiently carried out by the traditional communication buses without serious bottleneck issues, or point-to-point communication without serious space and energy waste. The cost of computation, which used to be more expensive than the cost of communication, is now in fact much cheaper and on-chip communication is becoming a major concern on manufacturers, since it encounters fundamental physical limitations. On-chip wires do not scale in the same manner as transistors do and the cost gap between computation and communication is getting bigger. The solution to this problem lies in the Network-On-Chip (NoC) architecture.

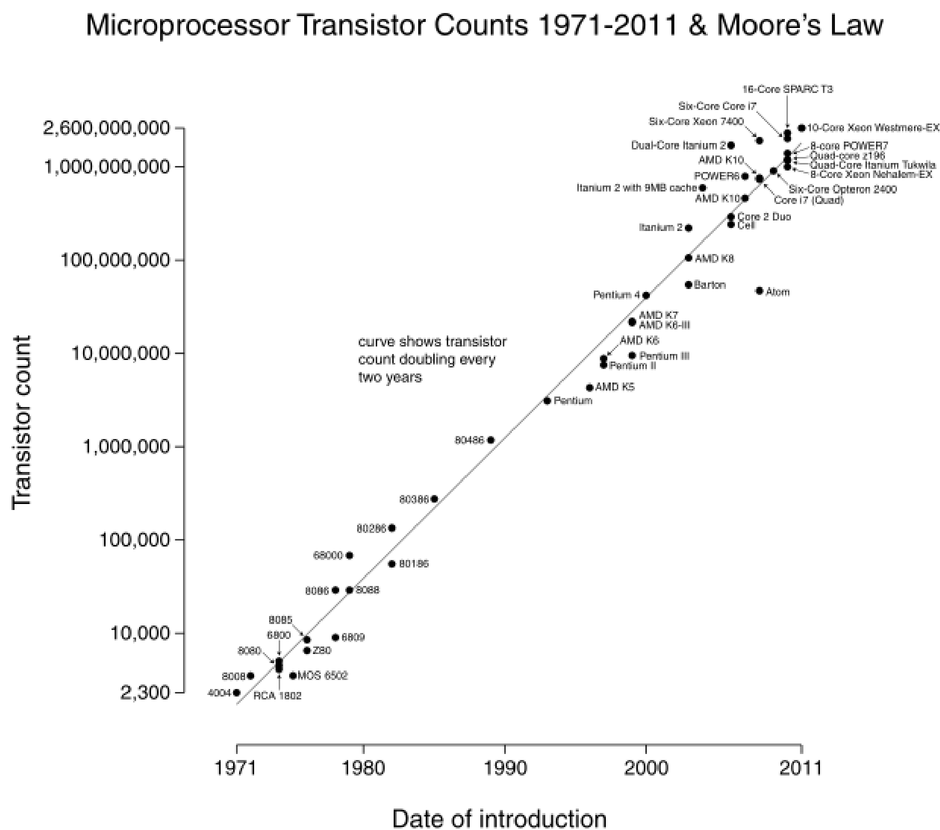


Figure 1.1: Moore's Law

1.2. Network-On-Chip

NoC is a new approach to the System-On-Chip (SoC) model and specifically Multi-Processor-System-On-Chip (MPSoC), which uses elements of Computer Networks for on-chip communication. A NoC consists of several Intellectual Property (IP) blocks, but instead of classical bus-based or point-to-point communications, a more general scheme is adapted, employing a grid of routing nodes spread across the chip. On every IP-block on the grid, a router is present, much like in computer networks, in charge for every data transaction from that node, even if it's not destined to the adjacent tile. Pros and cons of NoC over a data bus are shown on Table 1.1.

Bus Pros & Cons			Network Pros & Cons
Every unit attached adds parasitic capacitance, therefore electrical performance degrades with growth.	-	+	Only point-to-point one-way wires are used, for all network sizes, thus local performance is not degraded when scaling.
Bus timing is difficult in a deep submicron process.	-	+	Network wires can be pipelined because links are point-to-point.
Bus arbitration can become a bottleneck. The arbitration delay grows with the number of masters.	-	+	Routing decisions are distributed, if the network protocol is made non-central.
The bus arbiter is instance-specific.	-	+	The same router may be re-instantiated, for all network sizes.
Bus testability is problematic and slow.	-	+	Locally placed dedicated BIST is fast and offers good test coverage.
Bandwidth is limited and shared by all units attached.	-	+	Aggregated bandwidth scales with the network size.
Bus latency is wire-speed once arbiter has granted control.	+	-	Internal network contention may cause a latency.
Any bus is almost directly compatible with most available IPs, including software running on CPUs.	+	-	Bus-oriented IPs need smart wrappers. Software needs clean synchronization in multiprocessor systems.
The concepts are simple and well understood.	+	-	System designers need reeducation for new concepts.

Table 1.1: Pros & Cons of Bus and Network for on-chip communication [1]

As shown on this table, the biggest advantage of a Network instead of a Bus is the fact that it scales much better as more Intellectual Property blocks are included on larger systems.

The Bus becomes a bottleneck for the system, and its arbitration and testability slows down the whole system and the problem worsens as the number of masters increases. On the other hand, the network offers distributed computation of the routing and pipelining on the network wires, thus decongesting otherwise communication heavy areas. The Cons of the network lie on the fact that the IP blocks used are designed for bus-oriented communication and thus need to be rendered able to communicate in a network. In a similar manner, designers need to adapt to new concepts as well, but this will only be an issue, as both new IP blocks will be oriented towards network communication and SoC designers will respond to the new technological needs.

On comparison to Computer Networks, the NoC consists of the following components:

- *Cores* are Intellectual Property (IP) blocks, usually processors of any kind, containing some local memory. Can also be referred as *tiles* of the NoC.
- *Network Adapters* implement the interface by which the *cores* connect to the NoC.
- *Routing Nodes* are components similar to the routers in Computer Networks. They are in charge of applying the chosen routing protocols.
- *Links* connect the routing nodes, thus providing communication between them, via one or more physical or logical channels.

The *Routing Nodes* and the *Links* of the NoC consist the network in which the cores are connected. An example of a 4x4 mesh topology NoC is shown in fig. 1.2.

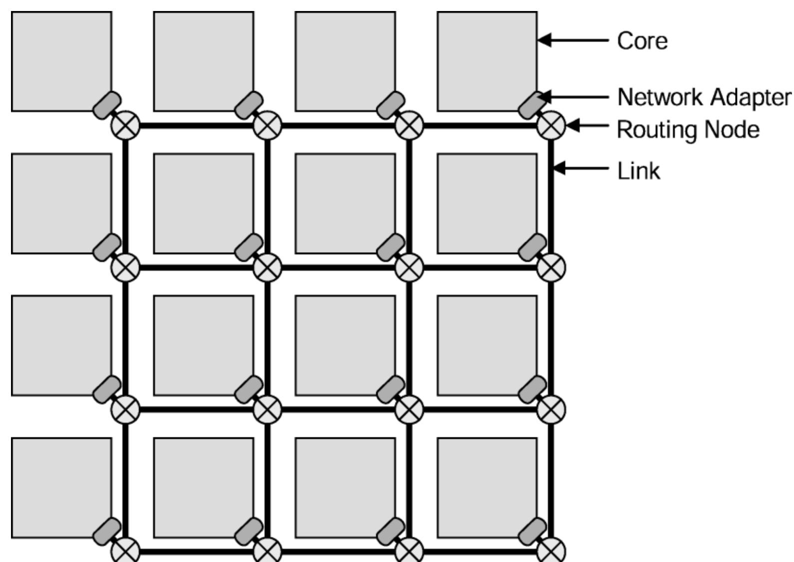


Figure 1.2: Example of a 4x4 NoC in mesh topology [1]

The cores communicate with each other as depicted in figure 1.3. The source core creates a message that needs to be delivered to the destination core. This message goes through the network adapter of the source core, which decides the destination, as the core itself isn't aware of the network, as will be made clear later on. Then the communicated data is forwarded to the core's routing node which, according to the destination, routes it towards any intermediate routing node, which does the same thing. Once the destination node is reached, the data goes in the opposite direction, from the router to the network interface and finally to the destination core.

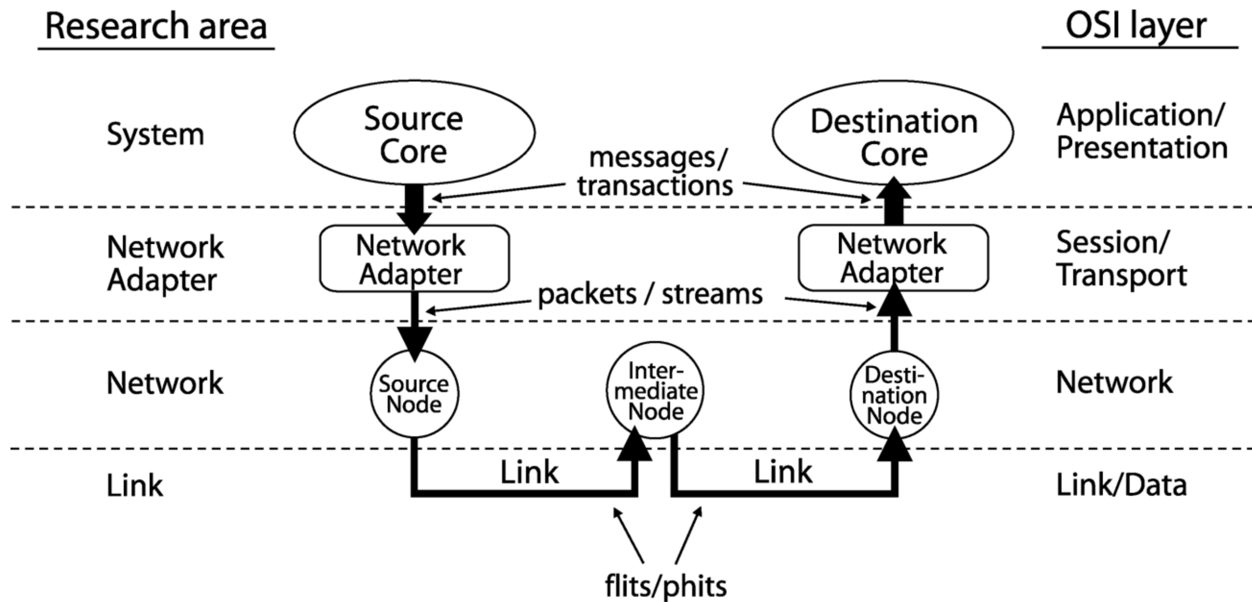


Figure 1.3: Communication between two cores.

1.2.1. Homogeneity and Granularity

As long as the type of cores on the NoC is concerned, the NoC can be characterized by its *homogeneity* and *granularity*. Thus, it can be *homogeneous* if all the cores belong on the same PE *type* and *heterogeneous* if more than one types exist on the same chip, just like the names suggest. For example, a homogeneous NoC can consist of processor tiles with local memory, and a heterogeneous one can include any of the following: processor-memory tiles, pure processor tiles, digital signal processors (DSP), memory tiles or even reconfigurable tiles like FPGAs. Furthermore, it can be *coarse* or *fine grained*, depending on the number of cores per surface. These options give NoCs increased flexibility and higher degree of variety over Computer Networks, which are mostly homogeneous and coarse grained. Examples of such NoCs are presented in fig. 1.4.

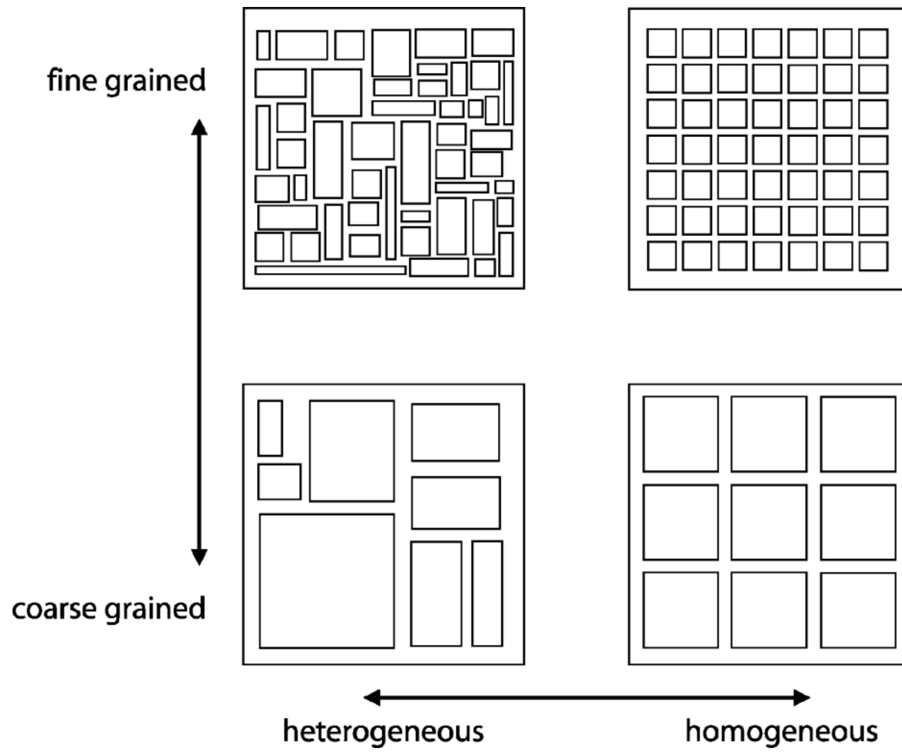


Figure 1.4: Effects of different degrees of homogeneity and granularity of system components. [1]

1.3. Network layers

A great advantage of NoCs lies in the readily accessible ideas of macro-networks, and the usage of nearly 50 years of research and work in the field of computer networking. That being said, based on ISO's Open System Interconnection (OSI) model, NoC's protocol stack comprises of the following 4 layers [1, 3]:

- The *System layer* involves solely the communication between the *cores* (conducted in *messages* or *transactions*), as well as their synchronization.
- The *Network Interface Layer* decouples the *cores* from the network and handles the end-to-end flow control, encapsulating the *messages* of the cores into *packets* or *streams*, to be sent via the *network.* This is the first level that is network-aware.
- The *Network Layer* consists of the routing nodes, links etc. defining the *topology* and implementing the *protocol* and the node-to-node control.
- The lowest level in the model is the *Link Layer* that involves the physical connection between the routing nodes, and the synchronization needed.

The NoC protocol stack can be seen in fig. 1.5. Shown in this figure is also the correlation with the *Application Programming Interface* (*API*).

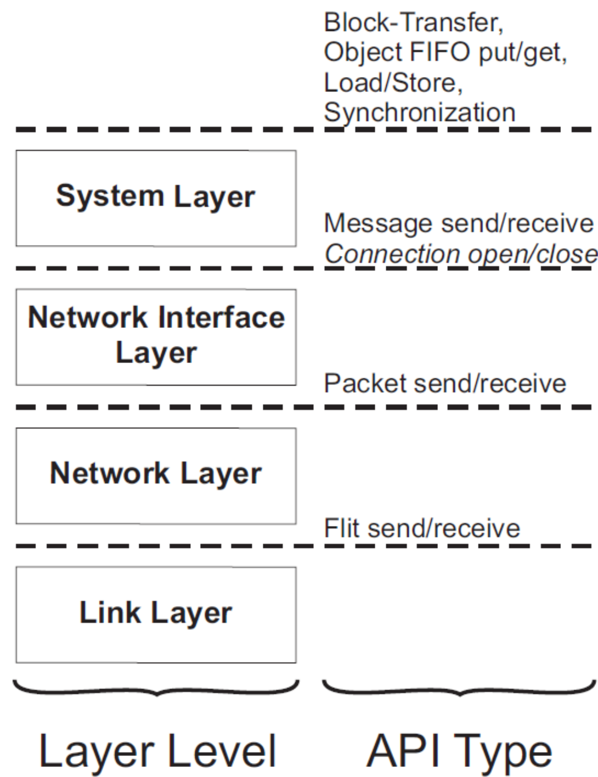


Figure 1.5: NoC layers and connection with the API [3]

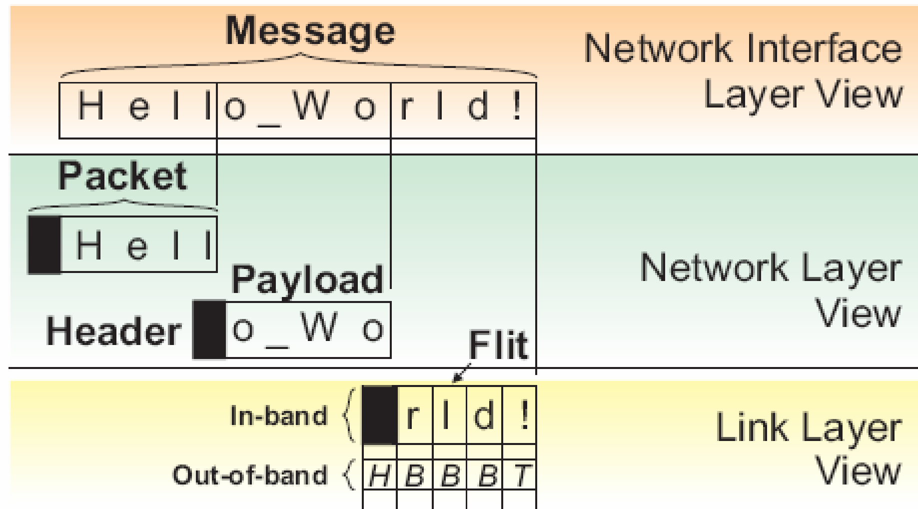


Figure 1.6: Decomposition of messages into packets and flits [3]

As depicted in figure 1.6, the data transactions on each layer take place with different data structures. The cores communicate with *messages*, which get decomposed into *packets* on the Network Interface Layer. The packets have a fixed size, and consist of a header containing routing information and the *payload*, which is the piece of the message they carry. When packets are ready to be transmitted in the link layer, they are further decomposed into pieces called *flits* or *phits,* which are physically transmitted through the wires. Flits are of different types, such as header (H), body (B), tail (T) and are transmitted out-of-band.

Following, each layer is further explained:

1.3.1. System Layer

The *System Layer* is the *Application Programming Interface* (*API*) that allows every node to communicate through the NoC. It encompasses applications (tasks or processes) and architecture (cores and network), and involves the data transactions and the synchronization between the *cores,* via *messages* or *transactions*. It also constitutes an interconnection between the IP-block and NoC's local protocol. This way, most of the network implementation details are hidden at this layer, introducing a level of abstraction, effectively hiding the hardware.

1.3.2. Network Interface Layer

The *Network Interface Layer* involves the NoC's *Network Adapters* (*NA*). Their purpose is to interconnect the adjacent *core* to the network, while decoupling them and ensuring the network remains hidden from the *system level*. Thus, they are responsible for *encapsulation/decapsulation, QoS management* and *NoC control services*. The *messages* or *transactions* of the *cores* are broken into *packets* that contain routing information, or *streams* which do not, but have a path setup before transmission.

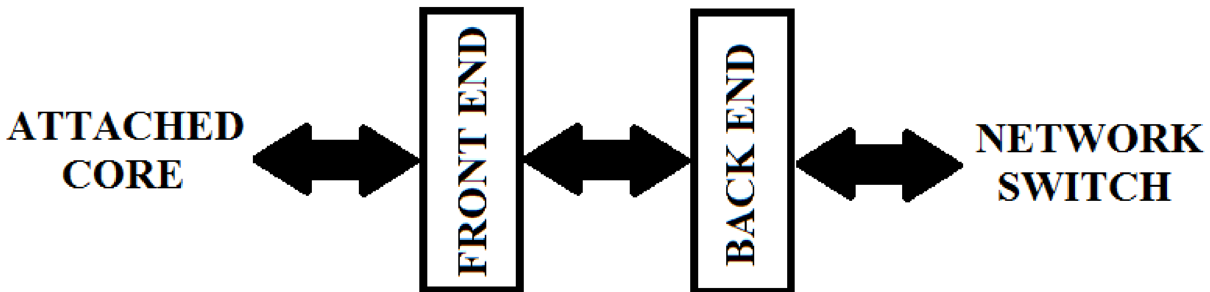


Figure 1.7: General network adapter [10]

As shown in figure 1.7, the *Network Adapter* implements two interfaces, the *core interface*, attached to the adjacent *core* and the *network interface* attached to the *network switch* or the *routing node*. The level of decoupling of the *core* from the *routing node* may vary. A high level of decoupling allows for easy reuse of cores, giving the designers great flexibility. On the other hand, a lower level of decoupling, that is a more network aware core, has the potential to make more optimal use of the network resources.

1.3.3. Network Layer

The purpose of the *Network Layer* is to pass portions of the *cores*' messages (called *flits* or *phits*) from a source core to a destination core. Ideally, the network should appear to its clients as simple point-to-point wires transporting data. In reality, routers (fig. 1.8) are used to forward data from one *core* to another.

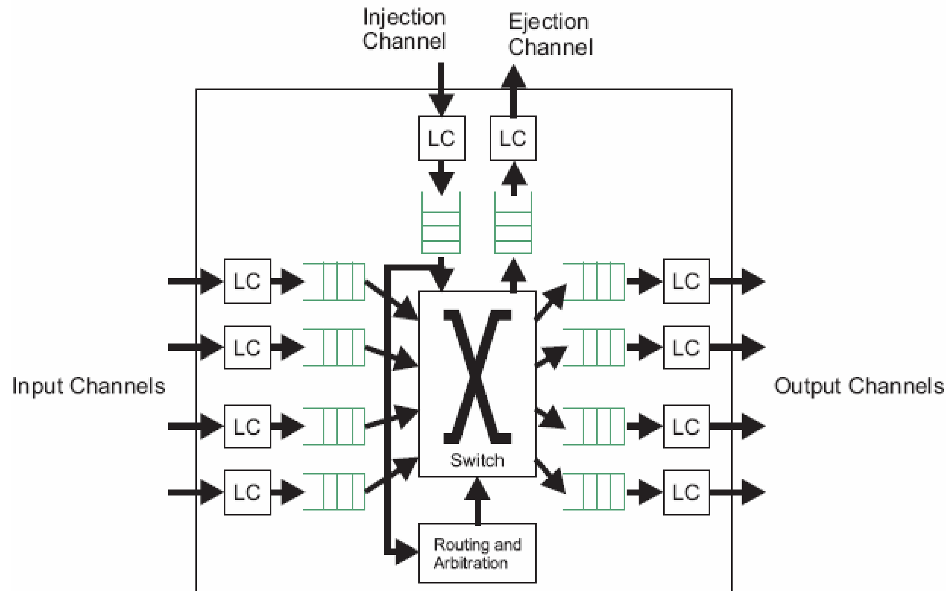


Figure 1.8: Typical structure of a NoC *router* [3]

The *network layer* is defined mainly by its *topology* and *routing protocol* used. The *topology* determines the layout of the connections between the nodes and the links. Topologies are characterized as *regular* and *irregular* ones. Some regular topologies are presented in fig. 1.9. The most used one is the mesh topology.

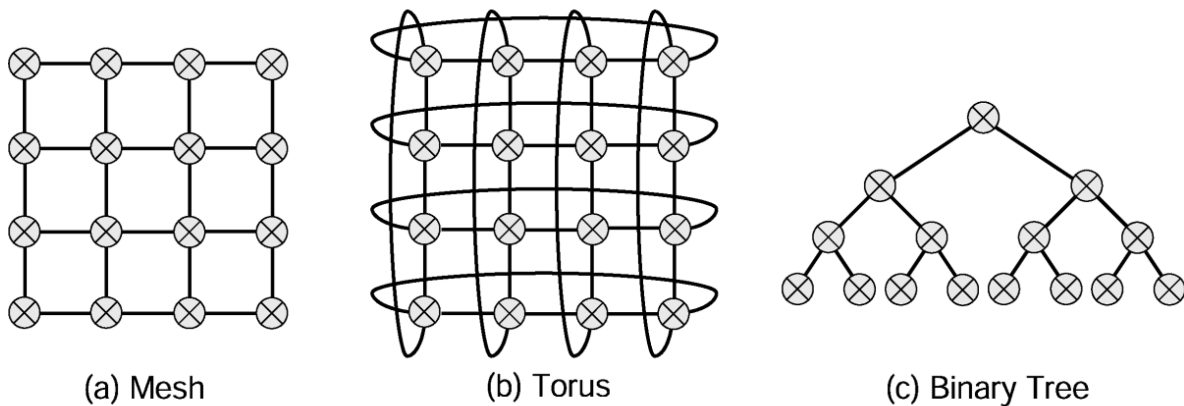


Figure 1.9: Regular forms of topologies [1]

The term irregular topologies is used to describe a free topology in which each node, including a router and one or more IP blocks, is possible to have a link with as many other nodes as desired by the designer. They can be created by either combining regular ones (fig1.10b), or using arbitrary connections between the nodes (fig1.10a), usually in order to take advantage of the

concept of clustering. They are intended for use in application specific purposes, contrary to regular topologies, that are intended mainly for general-purpose use.

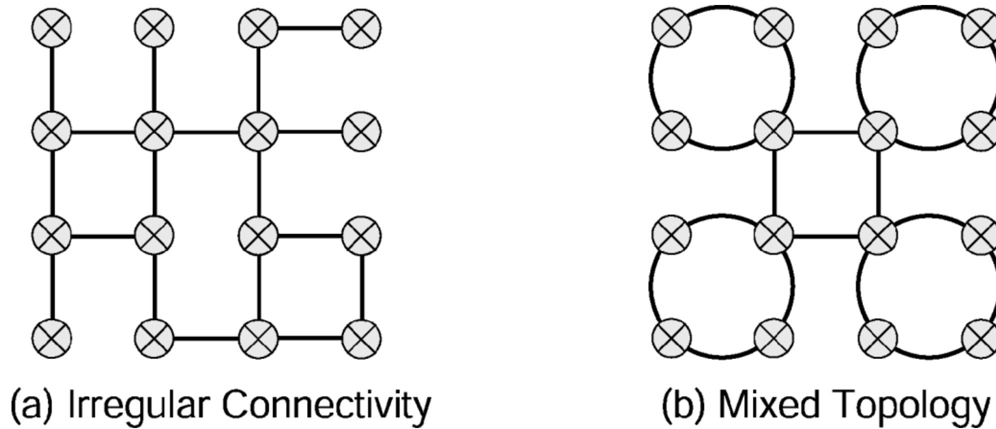


Figure 1.10: Irregular topologies

The *routing protocol* is the rule that determines the path the data will follow in the network from a source node to a destination node. The protocol can be classified as following:

- *Circuit switching* which involves the setup of a circuit from the source node to the destination node, that is reserved until the data transfer is over, or *packet switching* which involves the forwarding of *packets* (that contain data plus routing information) on a per-hop basis.
- *Connection oriented* where there are dedicated paths for each *data stream*, or *connectionless* where the path is determined dynamically for each *data packet*.
- *Deterministic routing* in which the path depends only on the source and destination tiles' coordinates, or *adaptive routing* where the routing path is determined on a per-hop basis according to the links' availability.
- *Minimal* or *non-minimal* whether or not the shortest path is always chosen.
- *Delay* or *loss models.* In the *delay model*, packets are never dropped, even if they are overdue, while in the *loss model* the packets can be dropped and be requested to be resent.
- *Central* or *distributed* control of the routing decisions.

The most common routing protocol that is used on NoC platforms is the *XY-routing* protocol. XY-routing is a dimension order routing protocol that suits well on networks using mesh or torus topologies, where the addresses of the routers are their Cartesian coordinates [4]. The protocol routes packets first in the x-axis (or horizontal direction) to the correct column and then in the y-axis (or vertical direction) to the receiver. An example on a 4x4 mesh-topology NoC can be seen in fig. 1.11.

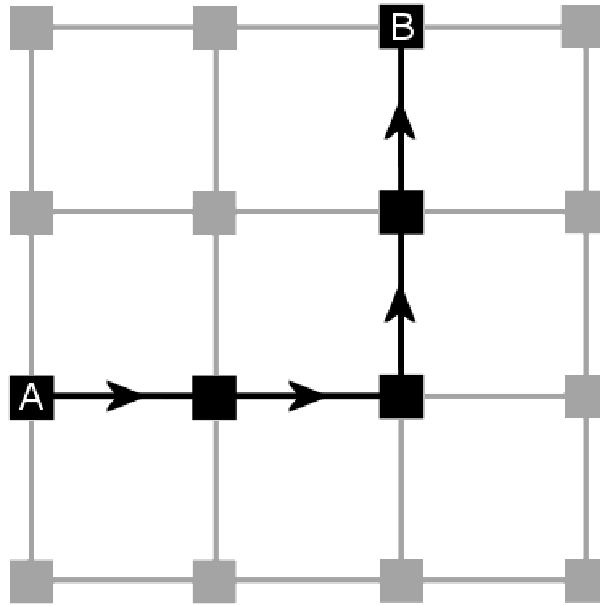


Figure 1.11: XY routing from router A to router B. [4]

1.3.4. Link Layer

The *link layer* deals with the point-to-point links between two neighboring *routing nodes*. These links consist of one or more physical or virtual channels. This layer abstracts many circuit-level and physical implementation details from the higher layers of the NoC to which it only exposes its atomic transaction the *flit* or *phit*. It deals with the following issues [3]:

- *Globally Asynchronous Locally Synchronous* (*GALS*) paradigm: With high clock frequency, the clock wavelength needs several cycles to traverse a whole chip. Therefore, synchronization throughout the entire chip is not possible. In order to cope with this problem, it is envisioned that there will be synchronous islands on a chip, connected via an asynchronous communication backbone.
- *Wire driving*: since the capacitive load is low, circuit techniques such as low-swing can be used to reduce the energy consumption on the wires.
- *Serialization*: Bit *serialization* of packets allows lowering the voltage of the link, hence lowering the energy consumption.

- *Bus encoding*: It has been proposed for on-chip communication in order to lower the power consumption per communicated bit, while simultaneously maintaining high speed and acceptable noise margin.
- *Wire pipelining*: Pipelining in the point-to-point wires between the *routing nodes* may be needed on high clock frequencies.
- *Flow control*: It is performed at the *link layer*, for instance in case the flow of data towards a saturated router needs to be suspended due to a full buffer. Moreover, *flow control* at the *link layer* involves the concept of virtual channels.

1.4. Run-Time Mapping

The concept of *NoC MPSoC* poses a new problem for the designer: Calculating a cost efficient *mapping* for a given application in a short amount of time. The application comprises of *tasks* being executed in parallel, on different *cores* of the NoC. The term *mapping* refers to the correspondence of each *task* on a different tile of the NoC to be executed, so that a *cost function* is minimized. An example of a mapping can be seen in fig. 1.12. It is of the outmost importance that the mapping happens in a short amount of time, so that the tasks can begin being executed, as soon as possible since a request has been made from the Operating System.

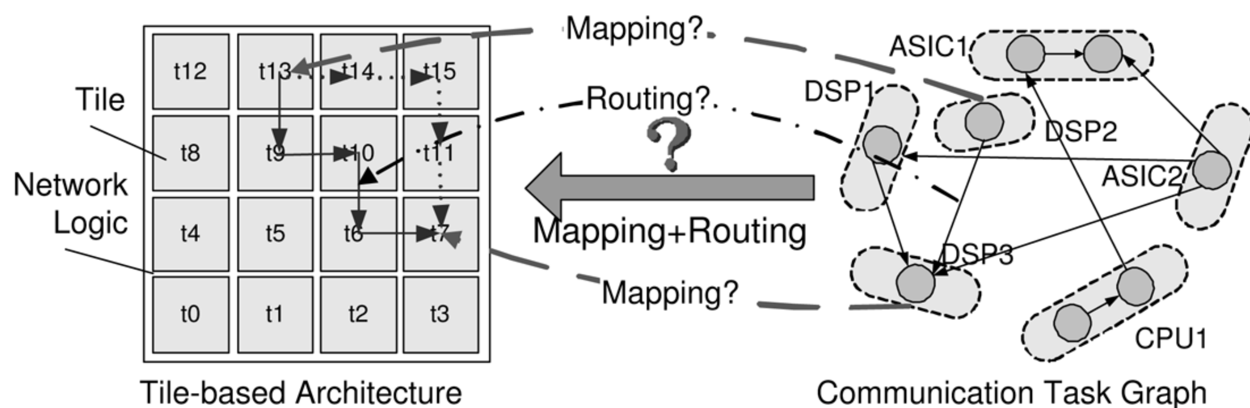


Figure 1.12: Illustration of the mapping/routing problem [11]

1.4.1. The Cost Function

The Cost Function may involve any performance metric needed to be minimized or maximized in order to achieve a good utilization of the platform's resources. That means that it can either be oriented towards *minimizing energy consumption*, or *maximizing performance*.

The first is wanted in embedded systems, where the power source and energy consumption of the system are a major concern for the designer. This is achieved in various ways, such as mapping tasks with heavy communication between them close to one another or, in case of heterogeneous platforms, mapping a task to a tile of the most energy efficient type.

On the other hand, a performance-oriented mapping tries to map every application on a tile of a type that it will be executed faster. Depending on the system's utility, having a balance between minimizing the energy consumption and maximizing the performance is often the problem in question.

1.4.2. Application Task Graph

In order to fully exploit the NoC's capabilities, the applications that run on it are divided in *tasks* that are executed in parallel. Tasks are portions of the application's code, usually with different resource requirements from each other. Each task is mapped on a different tile of the NoC, and inter-task communication takes place as part of the NoC's system layer. Due to this communication between the tasks, data dependencies occur, when one task utilizes data created in another task. Communication between tasks that are being executed in different rate can't be represented by data dependencies, since there isn't a one-to-one correlation between data derived from the source task and data needed in the destination task. A set of tasks with data dependencies is known as a *task graph* [5].

The *Application Task Graph* (*ATG*) is a directed graph G = (T, F), where T is the set of all tasks t_i of an application, and F is the set of data flows f_{ij} from task t_i to task t_j. An example is shown in fig. 1.13.

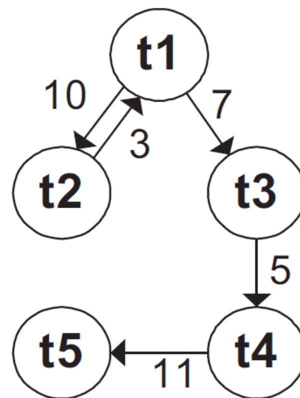


Figure 1.13: A simple ATG

The nodes of the task graph represent the tasks, while the flows represent some form of communication and data exchange between them. The weight of the flows can be any defining metric of the communication, for instance bandwidth required, latency, or cycles.

The ATG is the result of the application's profiling, and presents the information needed to describe the communication between the tasks. Thus, along with information about the tasks resource requirements and information about the NoC, the ATG is used as an input to the mapping algorithm.

1.4.3. Run-Time and Design-Time Mapping

Design-time decisions can often only cover certain scenarios and fail in efficiency when hard-to-predict system scenarios occur. This drives the development of run-time adaptive systems. Real-time applications are raising the challenge of unpredictability. This is an extremely difficult problem in the context of modern, dynamic, multiprocessor platforms which, while providing potentially high performance, make the task of timing prediction extremely difficult. The more complex a system grows the more it must be able to handle those situations efficiently.

Same principles apply for the decisions made in mapping. A run-time mapping is needed in order to move resource allocation out of design-time and its constraints. This way, a higher degree of flexibility is introduced on the platform. A design-time mapping just can't have the same amount of information and thus can't produce the best result. In fact, run-time mapping offers a number of advantages over design-time mapping. It offers the possibility:

- To adapt to the available resources. Those vary over time, due to applications running simultaneously. Run-time information can be incorporated to further reduce the cost of running an application.
- To enable unforeseeable upgrades after first product release time, e.g. new application and new or changing standards.
- To avoid defective parts of a SoC. Larger chips mean lower yield. The yield can be improved when the mapping algorithm is able to avoid faulty parts of the chip. Also aging can lead to faulty parts that are unforeseeable at design-time.

The only downside of a Run-Time mapping is the extra time it adds to the execution of an application, since it is executed between the request from the OS and the actual execution of the tasks. Hence, it is crucial that the mapping is calculated fast, so that it is transparent and doesn't burden the system.

1.4.4. Distributed and Centralized Mapping

Apart from defining the moment (run-time or design-time) the mapping occurs one must also define on which tiles the mapping algorithm will be executed. Therefore, mapping can be either *Centralized* or *Distributed* according to the strategy that selects cores to perform the mapping algorithm.

Centralized mapping utilizes one or a small set of cores, *Centralized Managers* (CM), to perform the mapping for every application that arrives. These cores then decide the mapping for the whole system. This mapping scheme may cause the following problems [6]:

- Larger volume of monitoring traffic. During the mapping, since it is performed on run-time, the Centralized Manager needs to collect data from the whole chip, which causes traffic on the wires, possibly stalling the execution of already running tasks.
- High computational cost to calculate the mapping for the whole chip at once.
- Single point of failure. If the Centralized Manager fails for some reason, the mapping can't be performed at all.
- The Centralized Manager becomes a point of *hot-spot* as every tile sends the status of the PE to it. This increases the chance of bottleneck issues around the manager.
- Scalability issues. As NoCs will grow in size, and more Processing Elements will be added, the computational effort of mapping and the traffic it will create will increase exponentially, thus rendering the computation very expensive and the scheme ineffective.

Distributed mapping on the other hand is designed to tackle these challenges. On this mapping scheme, the effort of the computation is distributed, as the name suggests, on several tiles across the chip, *Local Managers* (*LM*), and they may even change from one mapping to the next. This way, the problems of the problems of the Centralized mapping are solved as following:

- Less monitoring traffic. The Processing Elements only need to send the data to their closest Local Manager, and this way they travel less on the chip.
- The Local Managers only need to perform the mapping computation for the area of the chip they are responsible for, or for some designated tiles. This way the computation demanding problem is divided in less demanding ones.
- There are no issues of single point of failure or hot-spots, since the smaller portions of the computation can be performed on any tile.
- It scales very well with larger NoCs, since all that is needed is some more light-weight Local Managers, whose individual computation effort isn't increased.

Examples of centralized and distributed mapping are depicted in figure 1.14. In each of the two NoCs the manager or managers have been marked in the region they are responsible for. In figure 1.14a, the manager is responsible for mapping on the whole NoC. That means, that the manager has to communicate with all 15 tiles every time a new mapping is needed and consequently compute the best mapping taking into consideration the whole platform.

On the other hand, on the NoC depicted in figure 1.14b, the managers are responsible for 3 tiles each, and compute the mapping for 4 tiles at a time. This way both the data communication and the computational effort are reduced. In addition, more than one mappings could be computed simultaneously. The only downside is the synchronization needed between the managers, but it is trivial compared to the advantages.

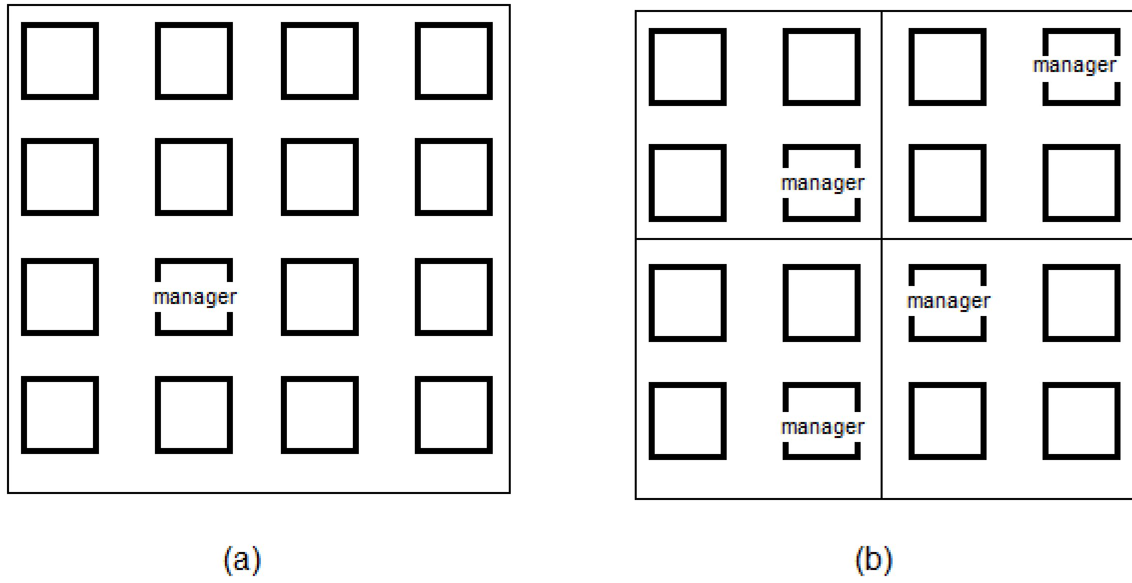


Figure 1.14:Examples of centralized and distributed mapping.

Chapter 2:
State-of-the-art Run-time mapping algorithms

2. State-of-the-art Run-time mapping algorithms

Research on run-time mapping for NoCs has been extensive and several algorithms have been published. In this chapter we briefly introduce four representative state-of-the-art works.

2.1. ADAM: Run-time Agent-based Distributed Application Mapping for on-chip Communication [6]

The authors of [6] propose a run-time distributed mapping scheme oriented to reducing the energy consumption and minimizing communication traffic in heterogeneous MPSoCs with NoC. The main idea is that in order to achieve the distributed computation of the mapping, the platform is partitioned in *virtual clusters* and computation of the mapping on each cluster is performed individually.

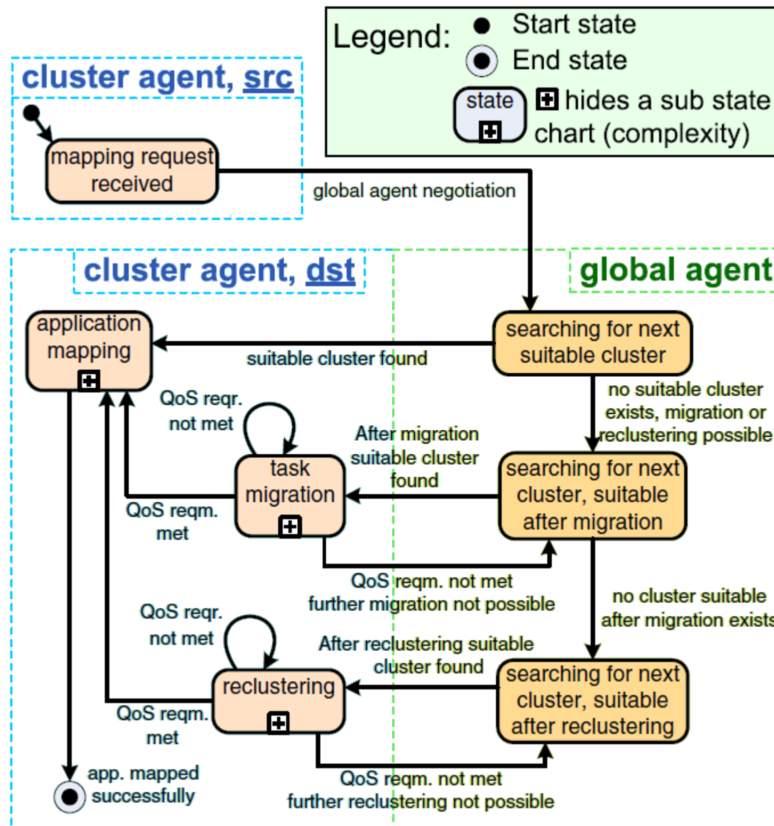


Fig. 2.1: Flow of the ADAM algorithm.

More specifically, a cluster is a subset of the set of tiles of the NoC. Its boundaries are not set and may change at any time, including more tiles, or excluding previously owned tiles. One of the cluster's tiles is selected to act as the *cluster agent.* An agent is a computational entity

which acts on behalf of others. The cluster agent specifically, is an agent that is responsible for mapping operations within its cluster.

Along with the cluster agents, there is another agent, the *Global Agent*. This particular agent stores the information for performing the mapping on any cluster. It is designed to be lightweight and easily movable, so that it can be hosted on any PE of the platform.

The flow of the ADAM algorithm is shown on fig. 2.1. When a new mapping request is received from any tile, the Cluster Agent of the tile's cluster communicates with the Global Agent, indicating the request. At this point, the Global Agent performs the *Suitable Cluster Negotiation Algorithm*, which finds a cluster capable to fit the whole application. The Suitable Cluster Negotiation Algorithm checks if there are enough free tiles in every PE type and resource requirement class for all the tasks in the application. In case no cluster is able, *task migration* occurs (taken from [7]), moving already running tasks to different tiles. If still no cluster is capable of hosting the application, the last resort is the *re-clustering* (fig 2.2), a process in which the clusters change in shape and possibly in number to better accommodate both the already running and the new applications.

After a cluster that can host the application has been found, that cluster's agent is responsible to perform the *Run-time Mapping algorithm* in which every task is appointed to a tile to be executed. This algorithm calculates the best tile for each task using a heuristics, checking the tile's position in the cluster (tiles near the center are preferred), the volume of communication on the tile before and after the mapping and the resource requirements for the task to run on any tile.

The great advantage of this mapping scheme lies on the concept of clustering and the low monitoring traffic, making it efficiently scalable on bigger NoCs.

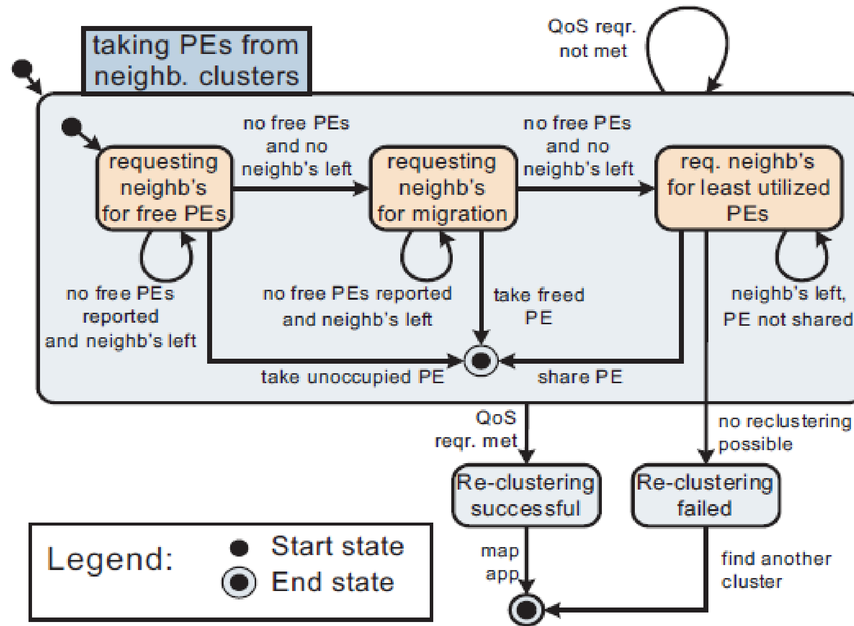


Figure 2.2: The re-clustering process of the ADAM algorithm

2.2. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles [7]

In this paper, the authors develop a Run-Time manager for heterogeneous NoCs containing fine grained *Reconfigurable Hardware Tiles*. Reconfigurable hardware is a type of Processing Elements, exhibiting its own distinct set of properties compared to traditional PEs (an example of Reconfigurable Hardware are FPGAs). It can be re-configured on run-time, according to the needs of the application, adding more flexibility to the NoC. These tiles are suited for computational intensive tasks, but can only accommodate a single task.

The proposed mapping algorithm, called *Resource Management Heuristic*, along with some add-ons for the reconfigurable hardware, is contained on the central Operating System, running on a designated tile, called the *Master PE*. The Master PE tile is responsible for assigning resources for both computation and communication to the different tasks (given as input in the form of an Application Task Graph that holds information about for both the properties of the tasks and the inter-task communication). The Operating System maintains a list of PE descriptors, keeping track of the computation resources of each tile, while the communication resources are maintained by means of an injection slot table that indicates when a task is allowed to inject messages onto a link of the NoC. In addition, every tile contains a Destination Lookup Table (DLT), used to resolve the location of its communication destinations. The Resource Management Heuristic follows the steps given below:

1. Calculate requested resource load.
2. Calculate task execution variance. In this step the sensitivity of every task to be mapped on any PE type is evaluated.
3. Calculate task communication weight.
4. Sort tasks according to mapping importance.
5. Sort PEs for most important task
    - Determine low communication – high performance tasks and their counterparts
    - Place together high communication tasks
6. Consider internal fragmentation of reconfigurable area. That means that sometimes the second best option is selected on step 7 if internal fragmentation of the reconfigurable tiles is too high.
7. Mapping the task to the best computing resource.

In case the mapping reaches a dead end, *backtracking* is used and if still no mapping is found *run-time migration*, *hierarchical configuration* or reduction of the *QoS* is used.

Hierarchical configuration of the tiles involves the use of softcore PEs instantiated on Reconfigurable Hardware tiles. This technique can improve the mapping performance when a task's binary isn't supported for execution on any of the NoC's other PE types or when it is more efficient communication-wise to map a task on a nearby Reconfigurable Hardware tile, rather than a further away PE tile.

In addition to the mapping algorithm, two *run-time task migration* schemes are proposed in this paper. It is defined as the relocation of an executing task from one tile to another. Task migration is used in case of a mapping failure, or whenever the user requirements change. It is considered that a migration can only occur in pre-defined points in a task's code, called migration points, in order to overcome architectural differences between different PE types in heterogeneous platforms. In order to maintain communication consistency two mechanisms are introduced:

- The *General Task Migration* mechanism.
- The *pipeline* mechanism.

The General Task Migration mechanism is described in figure 2.3. It is more efficient when moving a single task in order to e.g. resolve a mapping issue.

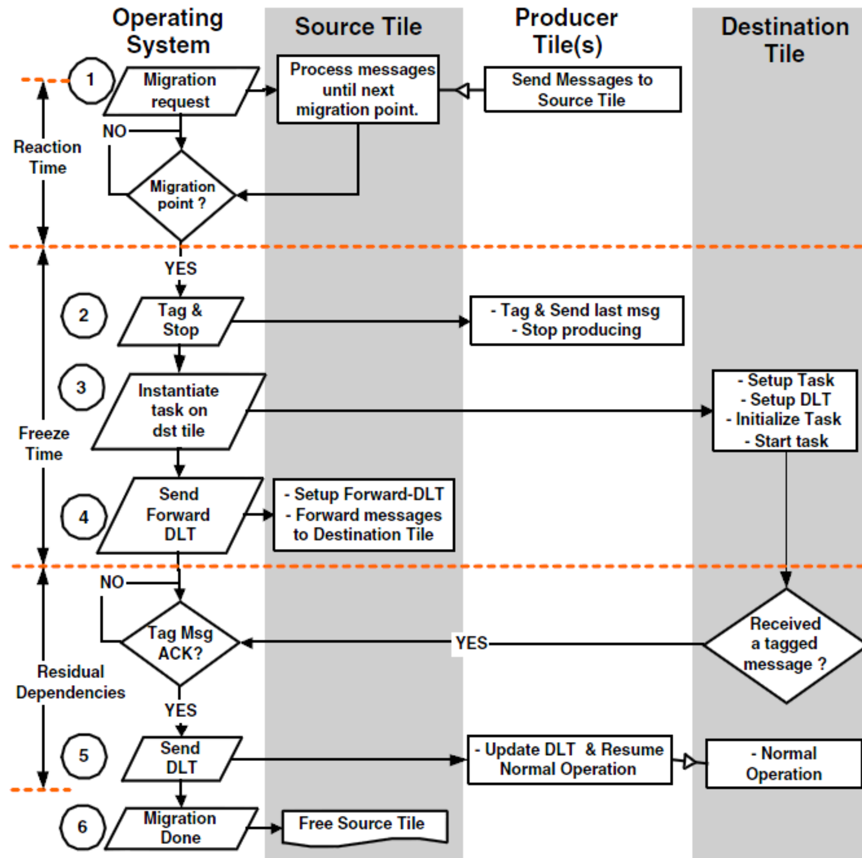


Figure 2.3: General Task Migration mechanism

The pipeline mechanism is based on the assumption that many algorithms are pipelined and contain *stateless points*. Stateless points are moments where new and independent data is put into the pipeline. This assumption allows a migration mechanism to move multiple pipelined tasks at once without being concerned about transferring task state. This mechanism is useful when new QoS requirements affect an application and tasks must be reallocated.

The mapping algorithm proposed in this paper, isn't the most effective possible, since it encounters the constraints of being centralized. Nevertheless, the migration mechanisms proposed can be very useful as parts of any run-time manager that uses the migration technique (like in [6]).

2.3. Incremental Run-time Application Mapping for Homogeneous NoCs with Multiple Voltage Levels [8]

This paper deals with the Run-time Mapping of Applications on Homogeneous NoCs. What makes this mapping scheme stand out is the prediction that the Processing Elements can

operate on *multiple Voltage Levels* (therefore multiple frequency levels), under different energy-performance trade-offs. The focus of this paper is not on determining the voltage island partitioning, and it is assumed that this is already determined on the platform. The mapping is also characterized *incremental*, meaning that not always the best solution is selected, in respect to better accommodating future applications that may occur, as opposed to a greedy algorithm that always chooses the best available solution.

The NoC platform is considered to consist of two separate networks, the *data network* where all data communication is carried out, and the *control network* where all the control signals pass through. There are separate networks for control and data, in order to make sure that data transmission does not interfere with the control messages of the Operating System.

The proposed mapping algorithm runs on a designated tile, called the *Global Manager*. This tile is responsible for making all the decisions for the mapping, thus making the mapping centralized, with all the subsequent disadvantages.

The input of the mapping algorithm is an Application Task Graph. It contains the set of tasks and some of their properties such as the worst-case scenario execution time and the minimum voltage that a tile can have in order to be able to execute them effectively. These properties have been obtained by means of *off-line partitioning*, in which some tasks may be profiled as *critical*, needing to be mapped in higher voltage tiles to be executed faster. The Task Graph also contains 2 weights for each edge, representing the *communication volume* (in bits) and the *bandwidth* (bits per sec) needed for the data flow. The mapping algorithm consists of two steps:

- Near convex region selection
- Node Allocation within the selected region

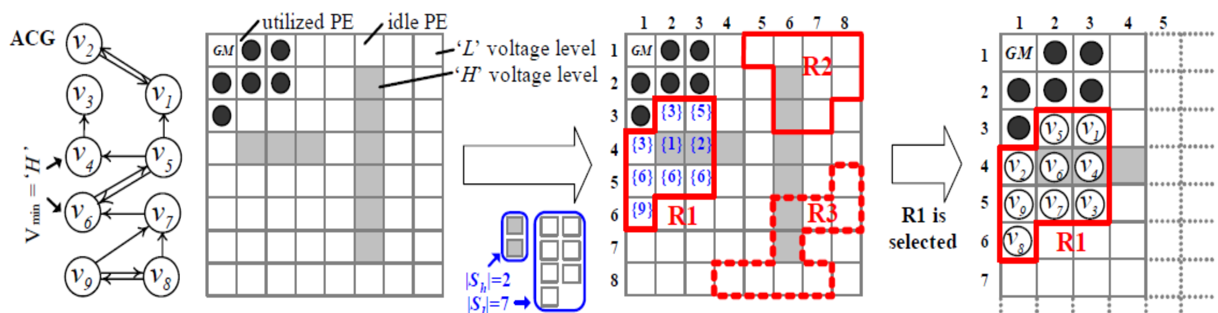


Figure 2.4: Incremental run-time mapping process

In the first step, a near convex (and contiguous if possible) region of the NoC is found, containing exactly as many tiles as needed by the application, with the appropriate voltage levels. Tiles are selected to be added to a region under two criteria concerning their position: their *dispersion factor* and their *centrifugal factor*, and of course the criterion of their voltage level. The dispersion factor is defined as the number of idling neighbors of the tile, with higher values

meaning that the tile is more prone to be selected, since it is likely to be later isolated. The centrifugal factor is defined as the Manhattan distance between a tile and a region's border. Hence, the lower the value of the centrifugal factor, the higher the probability that the tile will be added to the region, in order to preserve its contiguity.

Once a region that can host the application has been found, the algorithm moves to the next step of Node Allocation. The purpose of this step is to calculate the best tile from the selected region for every task to be executed on. For this, the tasks are sorted by their communication volume, and starting from the most communication-heavy one, the tiles that can host it are marked. When possible tiles have been determined for all tasks, starting from the head of the sorted set again, every task is assigned to the tile from the set of possible ones that minimizes the distance from the communicating tasks.

This mapping algorithm is executed in the Global Manager tile. That may cause serious scaling problems on larger NoCs, since it's a centralized mapping scheme. It is mentioned that on larger NoCs a hierarchical control mechanism should be applied. A variation of the algorithm for decongestion of the Global Manager would have a tile from each region calculating the Node Allocation step assigning tasks for all the tiles in the region, including itself, like a one-time Cluster Agent from [6]. This way, the algorithm would be more distributed and would scale much better. Lastly, an advantage of this algorithm is the fact that it is not limited to mesh topologies, but can easily be modified for many other topologies.

2.4 Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip [9]

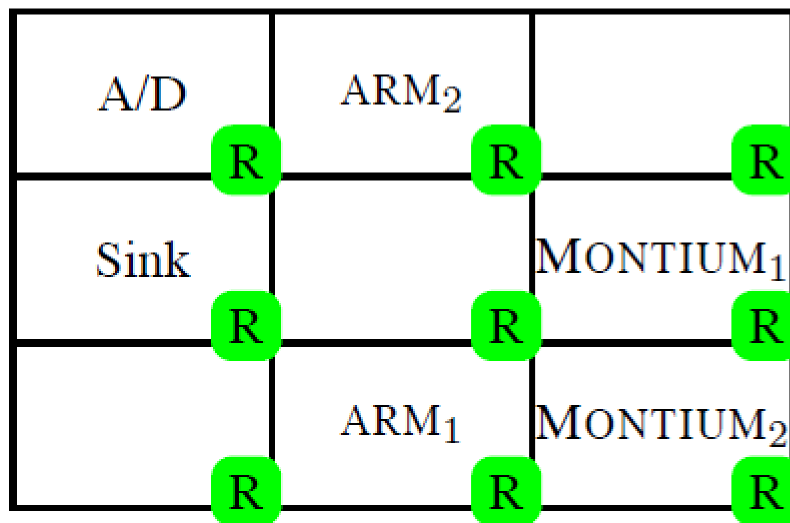


Figure 2.5: The Platform used in [9].

The authors of [9] propose a *Spatial Application Mapping scheme* for heterogeneous MPSoCs interconnected by means of a NoC, performed in run-time. It is intended to mapping *streaming DSP applications*, since, as noted, the concept of run-time mapping fits mainly to long running applications. The objective of the algorithm is to minimize the energy consumption for the execution of the streaming application, while meeting its QoS constraints.

The applications are considered to be described by *Cyclo-Static Data Flow graphs*, containing the *Worst-Case Execution Time* and *token production and consumption rates* for all different phases of execution of a task. In addition, it is considered that in order to be able to utilize heterogeneous MPSoCs efficiently, tasks can be implemented for any tile type. The example of the Fast-Fourier-Transformation algorithm is given, that can be executed on a DSP kernel, on an embedded ARM tile or a reconfigurable core.

The algorithm is described as a *hierarchical search with iterative refinement.* A mapping result can be characterized as *adequate* if all tasks can be executed on one of the platform's tile types, *adherent* when it is adequate and no has been assigned with more tasks that it can handle and *feasible* if it is adherent and the application's constraints are met. In order to reach a mapping the algorithm goes through these steps:

1. *Assign implementations to tasks*: Tasks are sorted by *desirability*, where desirability is defines as the difference between the cheapest assignment of a task to a tile type and the second cheapest. Starting with the most desired one, every task is assigned to the cheapest tile type that keeps the mapping adherent. After that, it is arbitrarily mapped to the first available tile of that type, so that a first concrete (greedy) mapping is reached.
2. *Assign processes to tiles*: On this step, iteratively, starting again from the most desired one, every task is removed from the tile it was assigned and it is attempted to be assigned on the best available tile of its tile type. Alternatively, in a local search type fashion, the task is swapped with another task and the best reassignment is performed on every iteration.
3. *Assign channels to paths*: The channels are sorted by decreasing throughput and for every channel a corresponding path is determined.
4. *Check application constraints*: The last step checks the QoS constraints. If any such constraint is violated, the mapping is *infeasible, feedback* is given to the earlier steps, and the mapping is performed again with the new data. If no QoS constraint is violated, the mapping is feasible and the algorithm ends.

A distinct characteristic of this mapping algorithm is the fact that it can be implemented either in a centralized manner, running on one core of the NoC, or in a distributed manner, with parts of it being executed on different tiles of the NoC. The difference in this algorithm from the previous ones is the concept of *feedback*. When a solution can't be found in anyone of the steps, the exact same algorithm is performed again iteratively, thus it has a low level of implementation difficulty on any Processing Element type.

2.5. Comparison Table

Following is table 2.1, summarizing the main characteristics of the presented algorithms.

	Centralized/ Distributed	Homogeneous or Heterogeneous system	Implements: RT mapping and/or Task migration	Implementation difficulty	Testing Platform	Experimentation on :	Flexibility with various size NoCs and/or applications	QoS taken into consideration	Application profiling	Minimization of :
[6]:ADAM Run-time Agent-based Distributed Application Mapping for on-chip Communication	Mainly distributed with centralized elements (global agent)	Heterogeneous	Both	High	Undefined NoC's of various sizes	A robot application, multi-media applications and task graphs	High flexibility, thanks to clustering	Yes	Yes, tasks are classified by type.	Energy consumption
[7]:Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles	Centralized	Heterogeneous	Both	Medium	StrongARM processor of a PDA connected to an FPGA containing a 3x3 NoC of the PE's	Task graph with random application load and random platform load.	Bottleneck problem on large NoC's. Flexible with applications thanks to RH add-ons	Yes, task load specification function takes under consideration the user requirements	Yes, requested resource load and weights are calculated for each task	Internal fragmentation on reconfigurable tiles
[8]:Incremental run-time application mapping for homogeneous NoCs with multiple voltage levels	Centralized	Homogeneous	RT mapping	Low	6x6 NoC of AMD ElanSC520, AMD K6-2E and one MicroBlaze core	Synthetic Benchmarks	Bottleneck issues on large NoC's, good scaling with Application size	Yes, some tasks are considered critical and have tighter deadlines	Yes, critical tasks exist	Communication energy consumption
[9]:Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip (MPSOC)	Not specified	Heterogeneous	RT mapping	Medium	Hypothetical NoC consisting of ARM and Montium tiles.	HIPERLAN/2 receiver	Scales well with NoC size, but not with Application size (many iterations)	Yes	Yes, on design time	Energy consumption

Table 2.1: The main characteristics of the 4 mapping algorithms.

Chapter 3:
Run-time Mapping (RTM) Algorithms

3.1 Main Idea behind the Run-Time Mapping (RTM) Algorithms

In this chapter the proposed Run-Time Mapping (RTM) algorithms of this work are presented. Two algorithms that share the main idea are proposed. They both are *Distributed Run-Time Spatial Mapping Algorithms,* the first one developed for *homogeneous* MP-SoCs and the second one developed for *heterogeneous* MP-SoCs. From now on, with the term RTM algorithm we refer to the main idea behind both algorithms, and the terms *homogeneous* or *heterogeneous* are used to distinguish between the two of them.

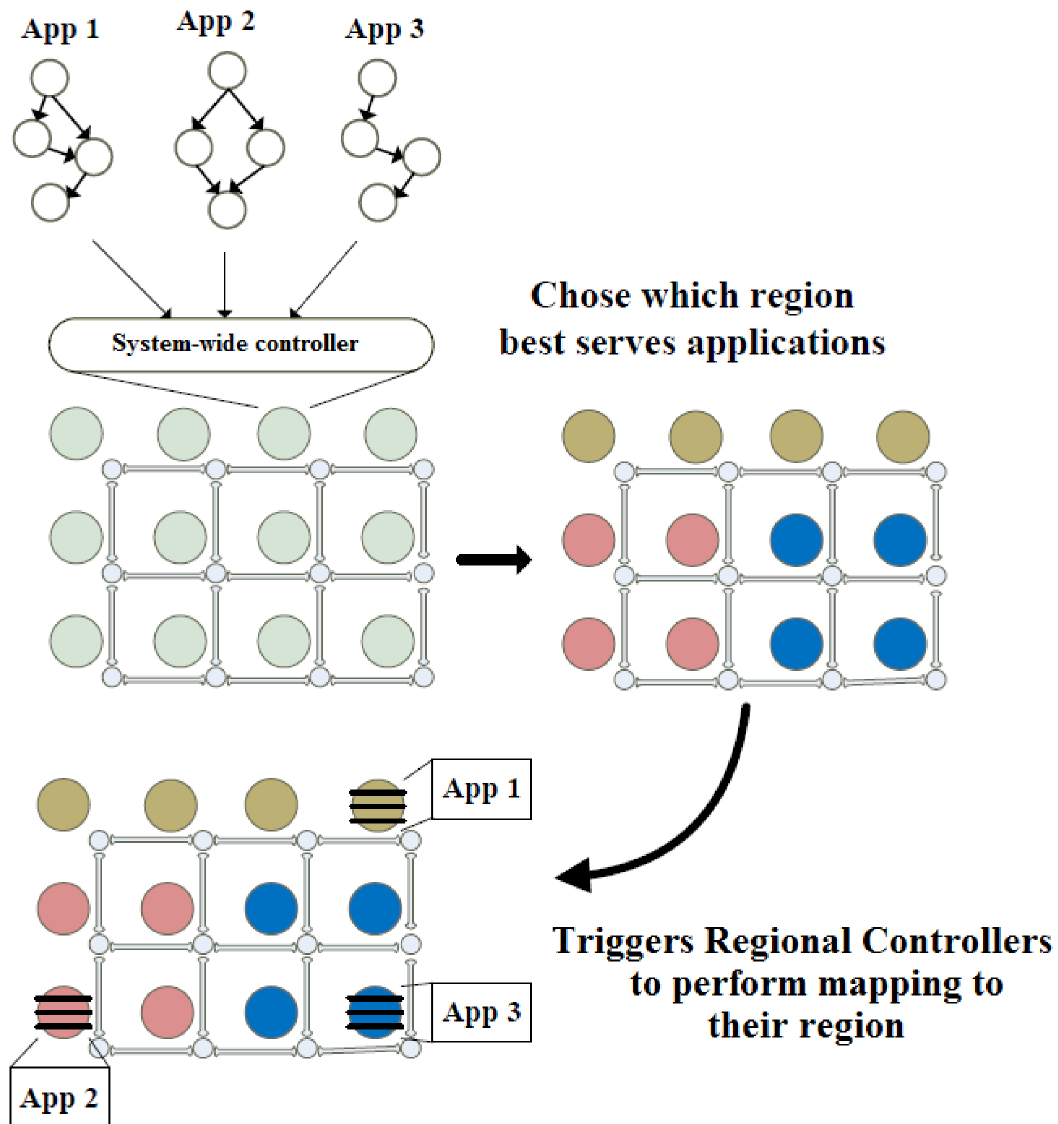


Figure 3.1: Main Idea of the RTM algorithm

The goal of the RTM algorithm, is the computation of an efficient mapping that minimizes the energy consumption from the execution of any application. We want this computation to be as fast as possible and transparent to the system, in order not to interfere with the execution of the algorithm.

An example of the implemented RTM algorithm is presented in fig. 3.1. The mapping is carried out in a distributed manner. In order to achieve that, the platform is partitioned in *regions,* i.e. subsets of the set of all the tiles on the NoC. These regions have no fixed boundaries, and can be reshaped, created or abolished when necessary. The manner in which the partitioning is performed is different in homogeneous and heterogeneous platforms, as will be shown later on.

Every new application mapping request is processed firstly by a designated tile, where the *System-Wide Controller* (*SWD*) task is being executed. This task is a lightweight piece of code, implemented for every type of Processing Element on the NoC in case of a heterogeneous platform, so that any core can assume the role of the controller, in order to keep the system protected from any single point of failure problems. This task's purpose is to find a region suitable to execute the new application, or take actions if the application can't be mapped for any reason. It holds easily transferable data for the whole NoC, based on which the resulting region is found. The collection of that data doesn't burden the whole platform, but only specific tiles as shown later.

In addition to the System-Wide Controller, there are some more designated tiles, one for each region, called *Regional Controllers* (*RC*). As the name suggests, these tiles are responsible for any action involving the mapping in their respective region. More specifically they are responsible for:

- Computing the mapping for the region for which the controller is responsible for.
- Collecting data for the region.
- Communicating and exchanging data with the System-Wide Controller.

In the same manner as the System-Wide Controller, the Regional Controllers are meant to be executable on any tile of the region, so that the platform's functionality doesn't depend on any single tile.

Once a region has been selected by the System-Wide Controller for the mapping of a new application, its Regional Controller is triggered and data describing the application is sent to it. Then the mapping is performed and its results are reported back to the System-Wide Controller (fig. 3.1).

The flow of the RTM algorithm for both homogeneous and heterogeneous platforms is shown on figure 3.2.

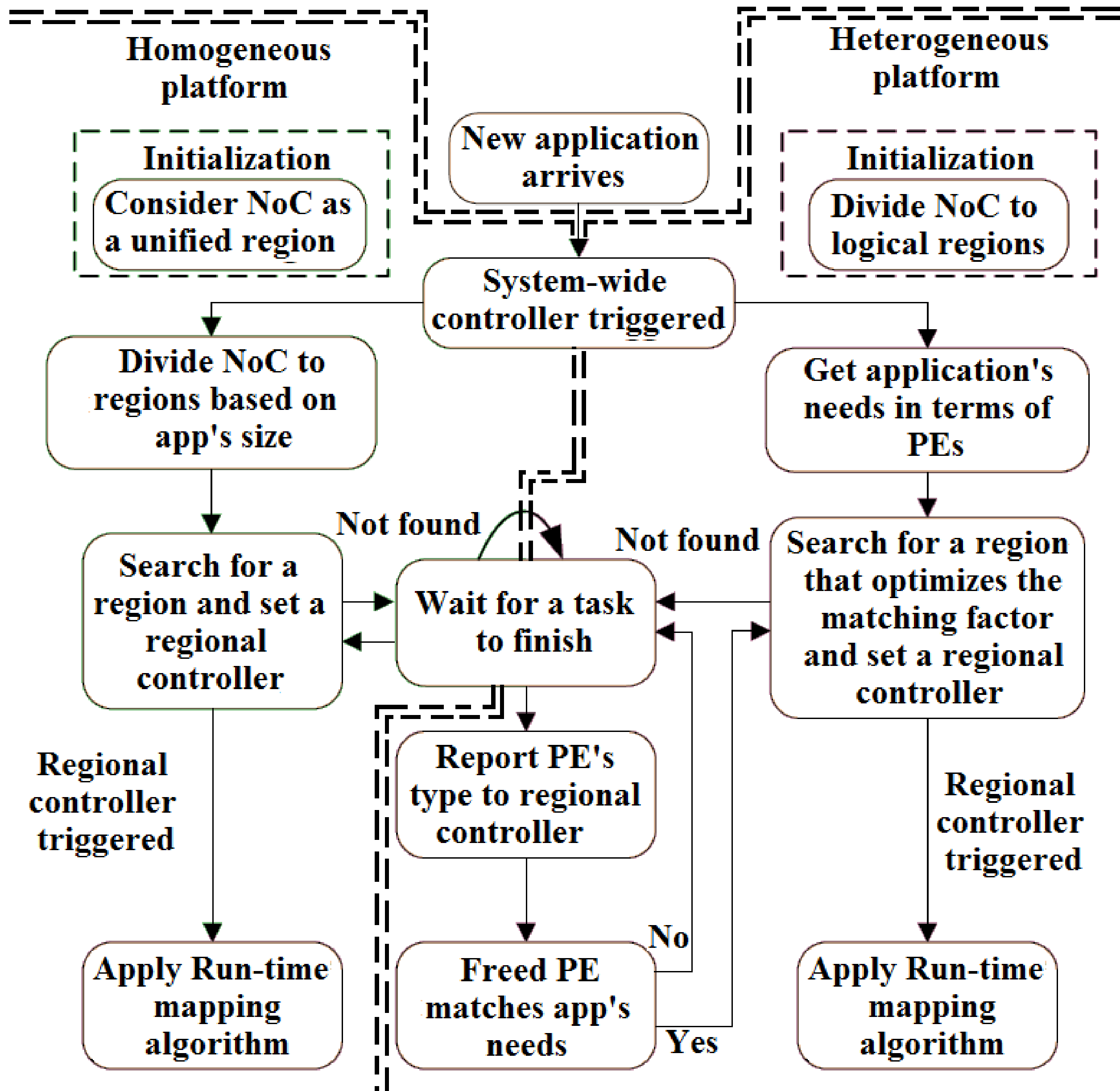


Figure 3.2: Flow of the RTM algorithm

3.2. Run-Time Mapping (RTM) on a homogeneous platform

The first of the proposed algorithms is intended to be used in homogeneous platforms, i.e. platforms with only one type of Processing Element tiles. That attribute makes the computation of the mapping easier, since there is no need to determine the most energy-efficient Processing Element type for each task. Thus, the most efficient mapping is derived mainly from minimizing the energy consumed by the inter-task communication.

3.2.1. Definitions

Definitions necessary to explain the RTM algorithm for homogeneous platforms are described in the following:

- The Application Task Graph (ATG) is used to capture the traffic flow characteristics. The ATG $G(T, D)$ is a directed acyclic graph, where each vertex $t_i \in T$ represents a computational module in the application. Each directed arc $d_{i,j} \in D$ between tasks t_i and t_j characterizes data and communication dependencies. Each $d_{i,j}$ has an associated value $b(d_{i,j})$, which stands for the communication volume exchanged between tasks t_i and t_j.
- A many-core platform's *topology* and *communication infrastructure* can be uniquely described by a strongly connected directed graph $A(I, N)$. The set of vertices N is composed of two mutually exclusive subsets N_{PE} and N_C containing the platform's Processing Elements and the platform's on chip interconnection elements (such as routers in Network-on-Chip technology) respectively. The set of edges I contains the interconnection information (both physical and virtual) for the N set.
- M_{PE} is the set of the *mapped* (*occupied*) *cores*. We also define a mapping function *map:* $T \rightarrow N_{PE}$ that maps the application's tasks (T set) to the available PEs (N_{PE} set). Let the set of unmapped nodes $\overline{\mathrm{M_{PE}}}$ such as $pe \in \overline{M_{PE}}$ if $pe \notin M_{PE}$. From our definition it follows that: $M_{PE} \cup \overline{\mathrm{M_{PE}}} = \emptyset$.
- R is the set that defines the logical regions on the platform. It is composed of k ($k \geq 1$) subsets $R_1, R_2, .., R_i, .., R_k$ called the $regions$ of the NoC.
- $M_{Ri}[]$ is a list that defines the one to one result of the map mapping function in the R_i region.
- A region R_i is considered occupied if $\exists pe_i \in R_i : pe_i \in M_{PE}$, that is, at least one of the tiles it contains is occupied.

3.2.2. The Run-time mapping algorithm for homogeneous NoCs

Algorithm 1: Run-time mapping for Homogeneous Platforms

	//Step 1: Check availability
1:	If $\lvert T \rvert \leq \overline{M_{PE}}$
2:	define new $R_i \in R \mid \forall pe_i \in R_i, pe_i \in \overline{M_{PE}}$
3:	signal(R_i)
4:	jump(Step 2)
5:	Else
6:	wait() //for a task to release its PE
7:	jump(Step 1)
	//Step 2: Run time mapping procedure
8:	sort $d_{i,j}$ by $b(d_{i,j})$ descending
9:	$\forall d_{i,j} \in D$
10:	$\forall pe_i \in R_i$
11:	$src = \min\{F_{HOM}(d_i, pe_i)\}$ //equation 1
12:	$dst = \min\{F_{HOM}(d_i, pe_i)\}$
13:	$M_{PE}, M_{Ri}[] \leftarrow src$
14:	$M_{PE}, M_{Ri}[] \leftarrow dst$
	//Step 3: Swapping procedure
15:	$bestCost = F_{swap}\{M_{Ri}[]\}$ //equation 2
16:	$\forall t_i \in T$
17:	$\forall t_j \in T, t_j \neq t_i$
18:	If $(MD(t_i, t_j) \leq MAX_MANH_DIST)$
19:	swap(t_i, t_j)
20:	$tmpCost = F_{swap}\{M_{Ri}[]\}$
21:	If $tmpCost \leq bestCost$
22:	$bestCost = tempCost$
23:	$M_{Ri}[] = new\ M_{Ri}[]$
24:	Else
25:	$swap(t_i, t_j)$

When a new application mapping request is issued, the System-Wide Controller processes it firstly, and performs the Step 1. Since this is a homogeneous platform, the Controller only needs to make sure that there is at least one tile for each task of the application (line 1). If there aren't enough unoccupied tiles the application can't be mapped at that time, so the System-Wide Controller has to wait until it is signaled by a Regional Controller that a tile has finished executing its task (line 6).

If the new application fits in the platform, i.e. there are enough unoccupied tiles to execute all the tasks, the System-Wide Controller appoints a tile on the NoC to be a Regional Controller, and sends the request to it. This Regional Controller is responsible for the region consisting of the *unoccupied* tiles around it, whose Manhattan distance from the controller is less or equal to the $search_distance$ value, which is defined as:

$$search_distance = \begin{cases} \sqrt{|T|}\,, |T| < 9 \\ \sqrt{|T|} - 1\,, |T| \geq 9 \end{cases}$$

The $search_distance$ is not a fixed value and may be increased to include more *unoccupied* tiles, as will be shown later.

The Regional Controller runs Step 2 of the algorithm. First the arcs $d_{i,j}$ of the ATG are sorted by their $b(d_{i,j})$ value (line 8). Then starting from the one with the highest b, tiles from inside the respective region are found to execute the tasks involved in that particular arc, that is the source (t_i) and destination (t_j) tasks, if they are not mapped yet (lines 9-14). If no more unoccupied tiles are left in the region, the $search_distance$ is increased by 1, and the search for tiles is done in the newly added tiles.

The tile n that is selected to execute a task is the one that minimizes the cost function:

$$\mathrm{F_{HOM}(n)} = \mathrm{a}\left(\frac{1}{\#\mathrm{tiles_{region}}}\sum_{\mathrm{l} \in \mathrm{region}} \mathrm{MD(l,n)} + \mathrm{bw(n)}\right) + \mathrm{b}\left(\sum_{\mathrm{k} \in (\mathrm{region} \cap \mathrm{M_{PE}})} \left(\mathrm{b}\left(\mathrm{d_{n,k}}\right) * \mathrm{MD(n,k)}\right)\right) \quad (1)$$

where: $\#tiles_{region}$ is the number of tiles in the region,
$MD(i,j)$ is the manhattan distance between tiles i and j,
$bw(n)$ is the bandwidth used on tile n towards all directions,
a, b are weights.

The term $\frac{1}{\#\mathrm{tiles_{region}}}\sum_{\mathrm{l} \in \mathrm{region}} \mathrm{MD(l,n)}$ in the function is used because tiles closer to the center of the region should be preferred over others near the border. The term $\mathrm{bw(n)}$ is used, since we prefer tiles with low bandwidth usage, rather than overburdening tiles with already high bandwidth. The term $\sum_{\mathrm{k} \in (\mathrm{region} \cap \mathrm{M_{PE}})} \left(\mathrm{b}\left(\mathrm{d_{n,k}}\right) * \mathrm{MD(n,k)}\right)$ is the sum of products of the bandwidth and the manhattan distance of the task in question mapped in the tile n and the rest of the already mapped tasks on the same application.

After the initial mapping has been performed an iterative application node swapping process is employed in order to further reduce the total communication cost (Step 3, lines 15-25). During this process every mapped task swaps tiles with any other task of the same application that is within a radius predefined by the value MAX_MANH_DIST. If the mapping after the swap is less costly (equation 2) than the previous one, the swapping is kept, else it is reverted. The cost used for the swapping is:

$$F_{swap} = \sum_{i,j \in T} b\left(d_{i,j}\right) * MD(i,j) \quad (2)$$

3.2.3. Example of the execution of the RTM algorithm on a homogeneous platform

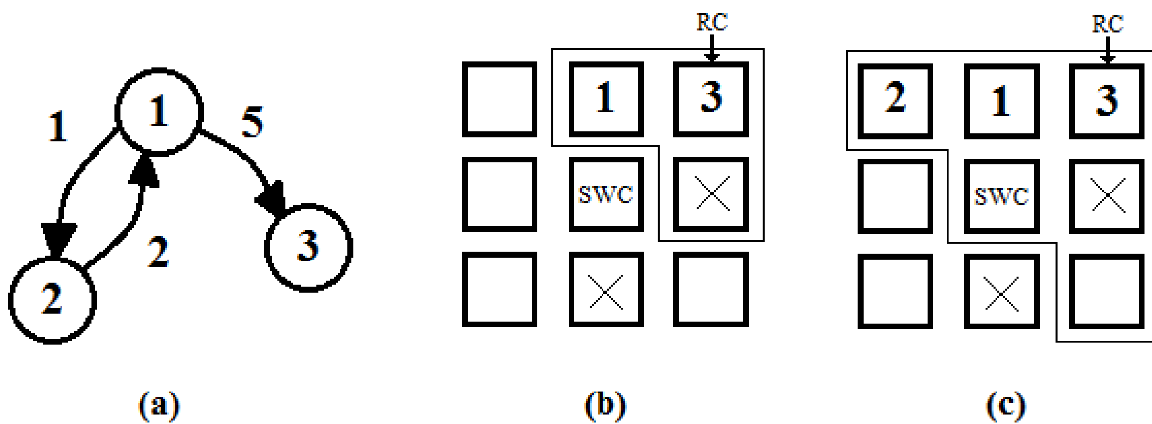


Figure 3.3: Example of the homogeneous RTM algorithm.
(a): ATG, (b): mapping of the first arc, (c): mapping of the other 2 arcs.

An example of the execution of the RTM algorithm on a 3x3 homogeneous NoC is shown on figure 3.3. The ATG consists of 3 tasks (3.3a), the System-Wide Controller is running on tile 1,1 and some tiles that are still running tasks from previous applications are marked with an x. On figure 3.3b the selected Regional Controller is shown and its initial region. There are 3 arcs on the ATG, with the costlier being the one from task 1 to task 3. Thus, task 3 is mapped first, followed by task 1 as shown on 3.3b. Next arc to be considered is the one from task 2 to task 1. Since, task 1 is already mapped, only task 2 needs to be mapped now. The region doesn't fit another task though, so the $search_distance$ value is increased, resulting in a wider region, and the final mapping is shown on 3.3c.

3.3. Run-Time Mapping (RTM) on a heterogeneous platform

Computing a mapping for a heterogeneous platform is more complex than the one for homogeneous platforms. This is due to the fact that the calculation of the most efficient Processing Element type for each task is needed, followed by the computation of a mapping that respects these preferences to types and in the same time minimizes the communication energy of any application's execution.

3.3.1. Additional definitions

In order to demonstrate the RTM algorithm for heterogeneous platforms the same definitions of terms as the homogeneous version are assumed, with the addition of new ones and enrichment of others as following:

- TP is the set of Processing Element types of the platform.
- The Application Task Graph (ATG) is the same with the one used for homogeneous platforms, with the only addition being $|TP|$ in number weights $W_j[t], t \in T,\ j \in TP$ on each vertex $t \in T$, one for each Processing Element type on the NoC. These weights represent the resources required for the execution of that particular task on every Processing Element type and are used to determine the preferred tile.
- The preferred Processing Element type for each task is denoted as $K[t], t \in T$. That is derived from the weights $W_j[t]$ of each task.
- $C[pe_i], \forall\, pe_i \in N_{PE}, C \in TP$ is the type of the type of the Processing Element pe_i.
- The $Matching\ Factor\ (MF)$ is a designer specified percentage value that defines on which types a task t_i can be mapped. For $MF = 100\%$ a task t_i can only be mapped to a tile of the $K[t_i]$ type, while for $MF = 0\%$ the task can be mapped on any type of Processing Element with respect to the preferences given by the $W_j[t_i]$ values. Different decisions for the MF value for the mapping results in different $M_{Ri}[]$ lists.
- R is again the set of regions on the platform, but in this case the following properties apply for the regions $R_1, R_2, .., R_i, .., R_k$: $\bigcap_{i=1}^{k} R_i = \emptyset$ and $\bigcup_{i=1}^{k} R_i = R$, which means that all Processing Elements of the NoC are part of a region, and that the regions are mutually exclusive.

3.3.2. The Algorithm for heterogeneous NoCs

A major difference between the RTM algorithm for heterogeneous and homogeneous NoCs is the way the regions are initialized. On homogeneous platforms during the initialization no regions existed, but were created for each new application, and abolished when the execution

finishes its execution. On heterogeneous platforms on the other hand, the NoC is partitioned in regions from the very beginning, with the selection of number and size of the regions left on the designer's judgment and Regional Controllers are appointed on these regions right away. Regions can be reshaped or created to better accommodate new applications, but they can only be abolished if they are an empty set, with respect to the properties:

$$\bigcap_{i=1}^{k} R_i = \emptyset \ and \ \bigcup_{i=1}^{k} R_i = R$$

Algorithm 2: Run-time mapping for Heterogeneous platform

//Initialization: best type calculation
1 : $\forall t_i \in T$
2 : calculate $K[t_i] : \min\left(W_{K[t_i]}[t_i]\right)$
//Step 1: Region selection step
3 : $\forall R_i \in R$
4 : If $(|T| \leq |\overline{M_{PE}}|)\&\&(\forall t_i \in T, \exists pe_i \in R_i : pe_i \in \overline{M_{R_i}}\ \&\&\ C[pe_i] = K[t_i])$
5 : select(R_i)
6 : jump(Step 5)
//Step 2: if the first matching doesn't yield a result
7 : store $T_{miss} \subset T : \forall t_i \in T_{miss} : \nexists pe_i \in R_i : pe_i \in \overline{M_{R_i}}\ \&\&\ C[pe_i] = K[t_i]$
8 : $\forall t_i \in T_{miss}$
9 : If $(\#iteration \leq MF * |TP| - 1)$
10: calculate new $K[t_i] : \min\left(W_{K[t_i]}[t_i]\right), K[t_i] \neq old\ K[t_i]$
11: repeat Step 1
//Step 3: if still no region is found, use *Region Reshaping*
12: $\forall unoccupied\ R_i \in R$
13: $\forall pe_i \in \overline{M_{PE}}, pe_i \notin R_i$
14: $\{R_i\} = \{R_i\} + pe_i$
15: repeat Steps 1,2
16: If R_i not selected
17: $\{R_i\} = \{R_i\} - pe_i$, restore(R_i)
//Step 4: no region was found, or all regions are occupied
18: create new $R_i = \emptyset \in R$
19: $\forall pe_i \in \overline{M_{PE}}$
20: $\{R_i\} = \{R_i\} + pe_i$
21: repeat Steps 1,2
22: If R_i not selected
23: restore previous R

```
24: wait() //for a task to release its PE
25: jump(Step 1)
        //Step 5: Run time mapping procedure
26: ∀K[t_k] ∈ G
27:     {S} ← d_{i,j} if (K[t_k] = K[t_i])||(K[t_k] = K[t_j])
28:     sort(S) by b(d_{i,j}) descending
29:     ∀d_{i,j} ∈ S
30:             ∀pe_i ∈ R_i
31:                     src = min{F_HET(d_i, pe_i)} //equation 4
32:                     dst = min{F_HET(d_i, pe_i)}
33:             M_PE, M_{R_i}[] ← src
34:             M_PE, M_{R_i}[] ← dst
        //Step 6: Swapping procedure
35: bestCost = F_swap{M_Ri[]} //equation 2
36: ∀t_i ∈ T
37:     ∀t_j ∈ T, t_j ≠ t_i
38:             If (MD(t_i, t_j) ≤ MAX_MANH_DIST) && (K(t_i) == K(t_j))
39:                     swap(t_i, t_j)
40:                     tmpCost = F_swap{M_Ri[]}
41:                     If tmpCost ≤ bestCost
42:                             bestCost = tempCost
43:                             M_Ri[] = new M_Ri[]
44:                     Else
45:                             swap(t_i, t_j)
```

When a new application mapping request is issued, in the same way as in homogeneous systems, the System-Wide Controller processes it firstly. Its purpose is to find a region capable of executing all the tasks of the new application. Prior to that, the controller calculates the preferences on the tasks on Processing Element types and assigns to every task the type with the minimum weight W_j (lines 1-2).

Next, the System-Wide Controller checks for every region R_i if the following equation applies (line 4):

$$|T_k| \leq |N_{PE,R_i,k}|, \forall k \in TP \quad (3)$$

where: $T_k \subset T: t_i \in T_k \; if \; K[t_i] = k, \;\; N_{PE,R_i,k} \subset N_{PE}: pe_i \in N_{PE,R_i,k} \; if \; pe_i \in R_i \; and \; C[pe_i] = k$

That means that a region is able to execute the application if it has enough tiles of every type to accommodate every task in the new application's graph that requires that particular type. For every region that the equation doesn't apply the set of tasks that can't be mapped is stored (line 7), since it is needed if no region is found. When the first region for which the equation 4 applies is found, its Regional Controller is signaled to perform the actual one-to-one mapping, as will be shown later.

In case no region is found on Step 1, before resorting to the communication heavy region reshaping, we try changing the tasks' most preferred type (Step 2) using *backtracking*. The tasks that couldn't be mapped previously get assigned to the next best Processing Element type (line 10), if the chosen Matching Factor allows it and the Step 1 is performed again to check if a region is found with the new data. Step 2 can be executed more than once, if no region is found in the previous iterations. The maximum number of iterations that can be achieved is determined by the value of the Matching Factor and equals: $MF * |TP| - 1$. For example on a NoC with 4 different Processing Element types, 100% MF means that the Step 2 will never be executed, while 50% MF means that the step can be executed once, so tasks can possibly be mapped to any of the two most efficient types.

Step 2 will probably not result to the most energy-efficient mapping, but at least the application will be mapped, and will not have to wait for another to end. This is where the concept of the *Matching Factor* is crucial. In case an application wants to wait for the most energy efficient mapping, the MF is chosen to have a value of 100% or generally high values. If on the other hand, energy isn't that important, but the application needs to be mapped fast, then the MF can be selected to have lower values.

However, there is still chance that none of the current regions can execute the new application, either because they are not large enough, or because many of their tiles are occupied, even due to the fact that the types don't match and the Matching Factor doesn't allow more type re-assignments to tasks. If this happens the *region reshaping* procedure is employed. As the name suggests, this procedure changes the regions in order to find a way to map the new application.

Initially, one after another every *unoccupied* region R_i 'borrows' all unoccupied tiles of the other NoCs temporarily (lines 13-14). Then Step 1 is performed again followed by Step 2 if needed. If a region R_i is selected at this point the run time mapping procedure that will be explained further on is performed, and all newly occupied borrowed tiles are permanently transferred to the region R_i while all still unoccupied borrowed tiles are returned to their previous regions.

Lastly, if all regions were occupied at the beginning of the region reshaping, then the System-Wide Controller creates a new region R_i that consists of all unoccupied tiles of the platform (lines 18-20). Once again, Step 1 is performed, followed by Step 2 if needed. If the new region is selected, its Regional Controller is signaled to perform the mapping procedure. If the new region still isn't capable of executing the new application, then the System-Wide Controller has to wait for tiles to finish the execution of earlier applications and free them.

After a region is found that can execute the new application, its Regional Controller is signaled and sent information about the application. At this point the System-Wide Controller's work is done and the Regional takes over and performs the run-time mapping procedure (Steps 5 and 6).

In the beginning for each Processing Element type K the flows $d_{i,j}$ that involve a task preferring this type, that is $K[t_i] = K$ or $K[t_j] = K$, are sorted by the respective values $b(d_{i,j})$. Then starting from the one with the highest b, tiles from inside the respective region are found to execute the tasks involved in that particular arc, that is the source (t_i) and destination (t_j) tasks, if they are not mapped yet (lines 31-32).

The tile n that is selected to execute a task t_i is the one that minimizes the cost function:

$$F_{HET} = F_{HOM} + \mathrm{a} * W_{K[t_i]}[t_i] \quad (4)$$

where F_{HOM} is the cost of the homogeneous run-time mapping (equation 1) and a is the weight from the same equation.

After the initial mapping has been performed an iterative application node swapping process is employed in the same way as the homogeneous platform (Step 6). During this process every mapped task swaps tiles with any other task of the same application that is within a radius predefined by the value MAX_MANH_DIST, and is mapped on a tile of the same Processing Element type. If the mapping after the swap is less costly than the previous one, the swapping is kept, else it is reverted. The cost used for the swapping is the same as the homogeneous case (equation 2).

3.3.3. Example of the execution of the RTM algorithm on a heterogeneous platform

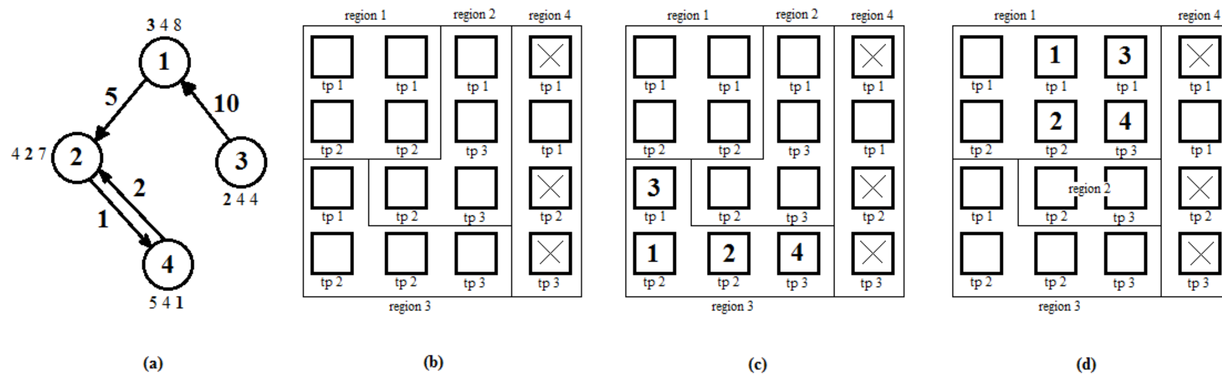


Figure 3.4: Example of the heterogeneous RTM algorithm.
(a): ATG, (b): The platform, (c): mapping with MF 50%, (d): mapping with MF 100%

An example of the execution of the Run-Time mapping algorithm on a 4x4 heterogeneous NoC is shown in figure 3.4. The ATG is shown in 3.4a, where the tasks, the communication between them and the weights $W_j[t]$ are noted. We assume a NoC with 3 different Processing Element types, as shown on 3.4b. Since $|TP| = 3$, there are 3 weights for each task.

From these weights the preferred types are derived, and thus this application would prefer a region with 2 type 1 tiles, 1 type 2 tile and 1 type 3 tile. The only region that has these tiles is region 4, but not all of them are free, since earlier tasks are still being executed. This is where the MF shows its significance. With an MF value of 50%, task 1 is allowed to be mapped on a tile of the second most preferred type, namely type 2. Thus, the mapping shown in 3.4c occurs. If the MF had a value of 100% on the other hand, the application wouldn't be able to get mapped on these regions, hence, the region reshaping procedure would be needed and the mapping in 3.4d would occur.

It is obvious that the mapping with MF 100% is more energy efficient since all tasks run on their most preferred Processing Element type, but the mapping needs more time to be executed, and in case more tiles were occupied, the new application would have to wait even more for region 4 to become unoccupied. Thus, the decision of the MF value is a tradeoff between the time needed for the mapping algorithm and the level the energy consumption is optimized.

Chapter 4:
Experimental results

4.1. Introduction

The Run-Time Mapping algorithms described in Chapter 3 have been implemented in C code using the MinGW port for the GCC compiler. We have performed extensive simulations of the behavior of several application benchmarks and random applications generated from TGFF [12] to validate our approach. The algorithm has been tested in the fields of execution speed and quality of the resulting mappings, as well as its behavior on scenarios of multiple application mapping requests over time.

4.2. The Application Platform

The Application Platform is a complex and full Multi-core NoC experimental platform presented in [2] (fig.4.1). It uses the LEON3 as the processor in each Processor-Memory node and uses the Nostrum NoC as the onchip network. Each PM node has a LEON3 processor (complete with I-Cache and D-Cache), a Data Management Engine (DME) [2] (fig. 4.2) as the network interface, plus a local memory. The LEON3 processor core is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The Nostrum NoC is a 2D mesh packet-switched network with configurable size, that uses the XY-routing protocol. It serves as a customizable platform.

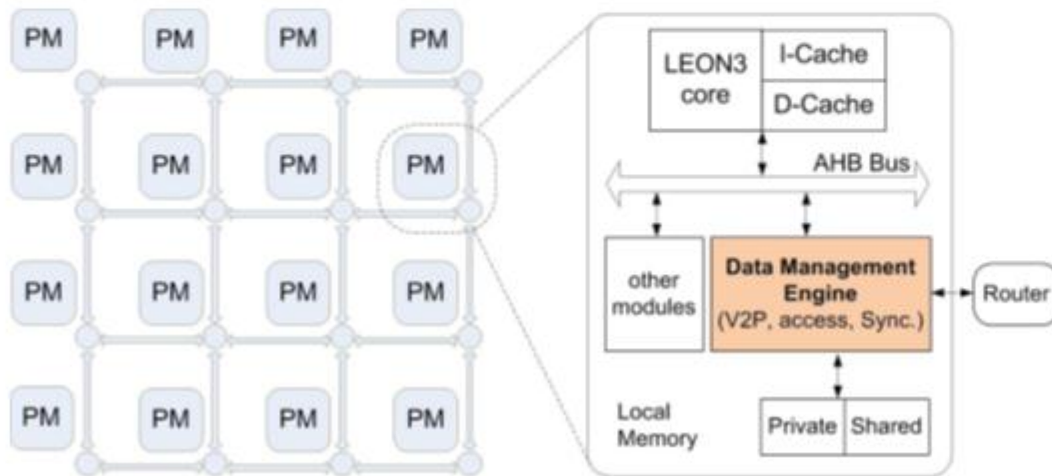


Figure 4.1: The application platform

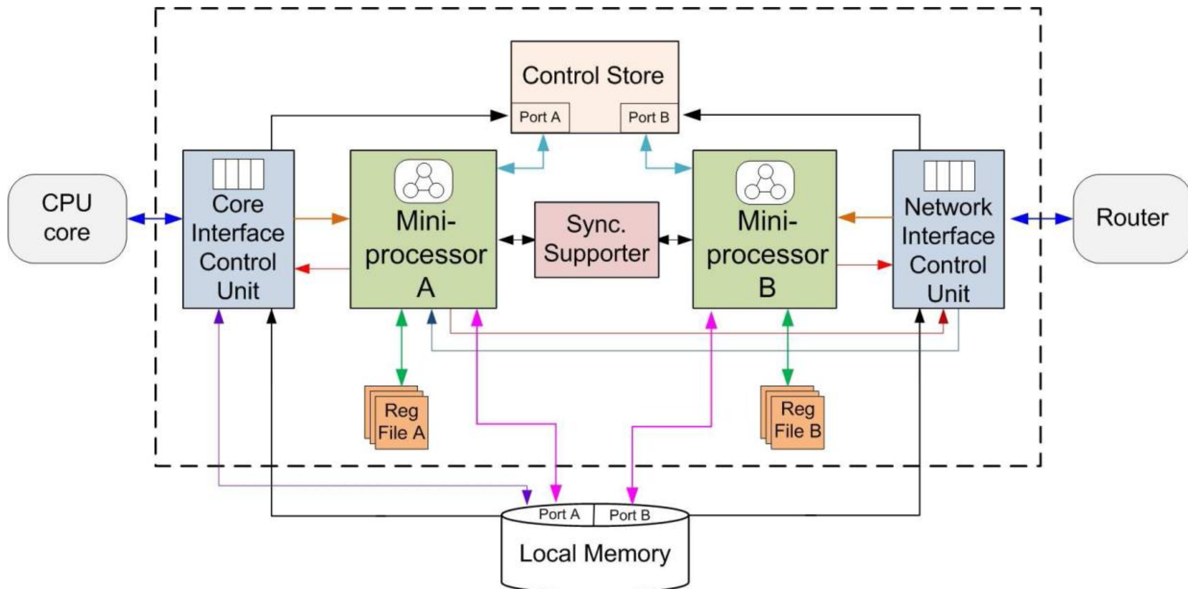
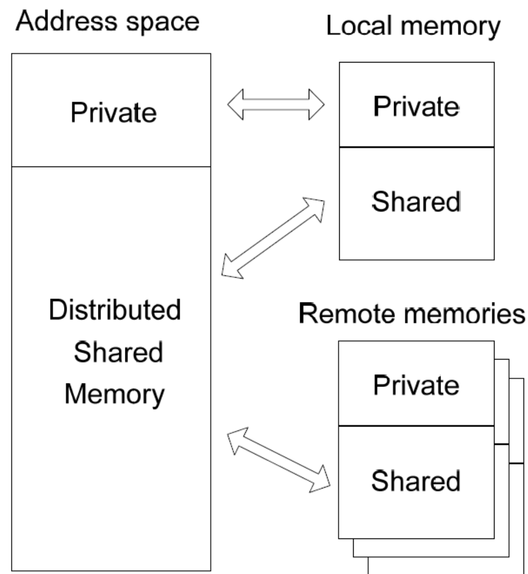


Figure 4.2: Architecture of the Data Management Engine



Memory Space	Information
0x40000000 ~ 0x401FFFFF	⇔ private memory
0x40200000 ~ 0x4021FFFF	⇔ shared memory #0
0x40220000 ~ 0x4023FFFF	⇔ shared memory #1
0x40240000 ~ 0x4025FFFF	⇔ shared memory #2
…	…

Figure 4.3: The global memory address space of each core.

Memories are distributed in each node and tightly integrated with processors. All local memories can logically form a single global memory address space (fig. 4.3).The local memory is partitioned into two parts: private and shared and two addressing schemes are introduced: physical addressing and logic (virtual) addressing. These two parts are separated by the *boundary address*, i.e. the first address of the shared memory (for example 0x40200000 in figure 4.3). The private memory is physical and can only be accessed by the local processor. All of shared memories are virtual, visible to all nodes and organized as a Distributed Shared Memory (DSM). The system uses a virtual-to-physical translation via virtual-to-physical (V2P) tables (fig. 4.4) to determine the addresses.

Shared memory No.	Node number
#0	Node(0,0)
#1	Node(0,1)
#2	Node(1,0)
#3	Node(1,1)
#4	Node(2,0)
#5	Node(2,1)

Figure 4.4: V2P translation table.

The V2P table is used as depicted in fig. 4.5. When a logic address needs to be resolved, if it is equal or higher than the boundary address of the core, the V2P table is accessed to determine on which core's shared memory the address belongs to, and then it is translated to that core's physical address.

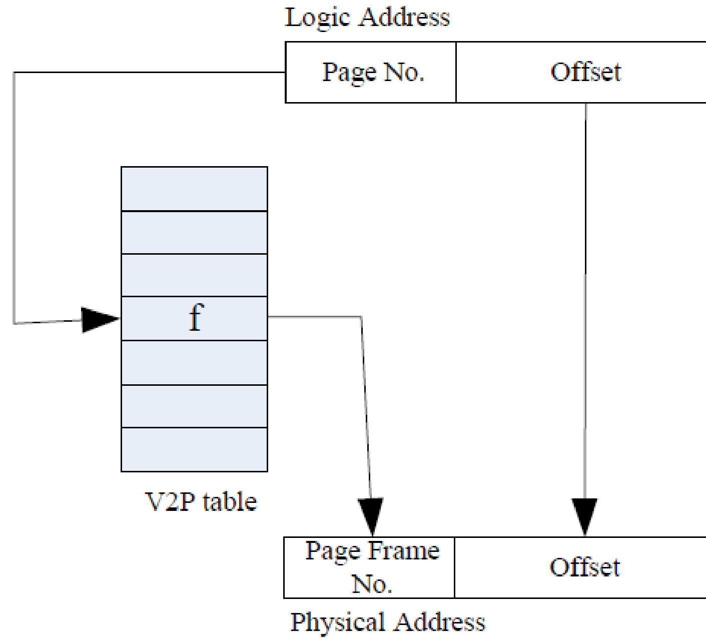


Figure 4.5: Usage of the V2P table.

The communication of cores inside the platform is done using message-passing instructions and by using the shared memory interface. Whenever there is a need for the System-Wide Controller to trigger another core, the hardware's synchronization *safe-lock memory mechanism* is used (fig4.6). Shared memory environment allows the easy use of such mechanisms. The lock is acquired by the system-wide controller and it propagates information to the shared memory. Then the lock is freed and the region controller loads the data from the memory and performs the required mapping operations. The execution of code on a Regional Controller is also possible with the usage of message passing instructions.

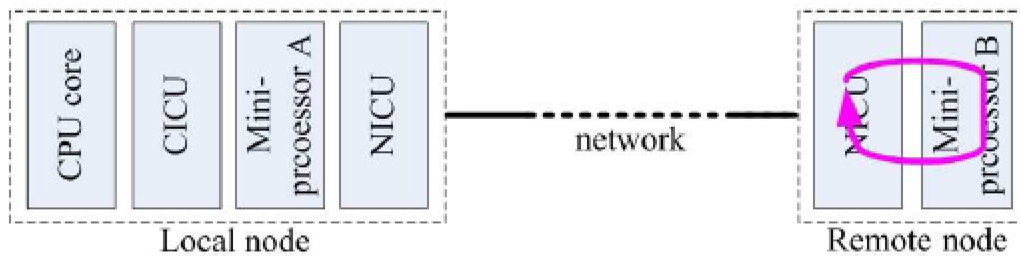


Figure 4.6: Representation of the spin-lock used

4.3. Experimental Results of the RTM algorithm for homogeneous platforms

Here we present the experimental results for the RTM algorithm on homogeneous platforms. The algorithm is compared to a state-of-the-art distributed run-time mapping algorithm and an exhaustive design-time algorithm, in terms of on-chip communication cost of the resulting mapping and computational effort of the algorithm itself.

4.3.1. TGFF generated applications

Tgff [12] is a user-controllable, general-purpose, pseudorandom task graph generator. It was used to create Application Task Graphs of various sizes, in order to experiment on the efficiency of the RTM algorithm. In order to test the effectiveness of our algorithm, it is tested against the *ADAM distributed run-time mapping* algorithm (presented in [6] and Chapter 2) and the exhaustive design-time mapping algorithm from [13].

In the following charts, where we compare the results of the algorithms, on the x-axis are the various NoC sizes the algorithms were tested, while on the y-axis are the computational cost of the resulting mapping or the cycles needed for the computation, both in logarithmic scale.

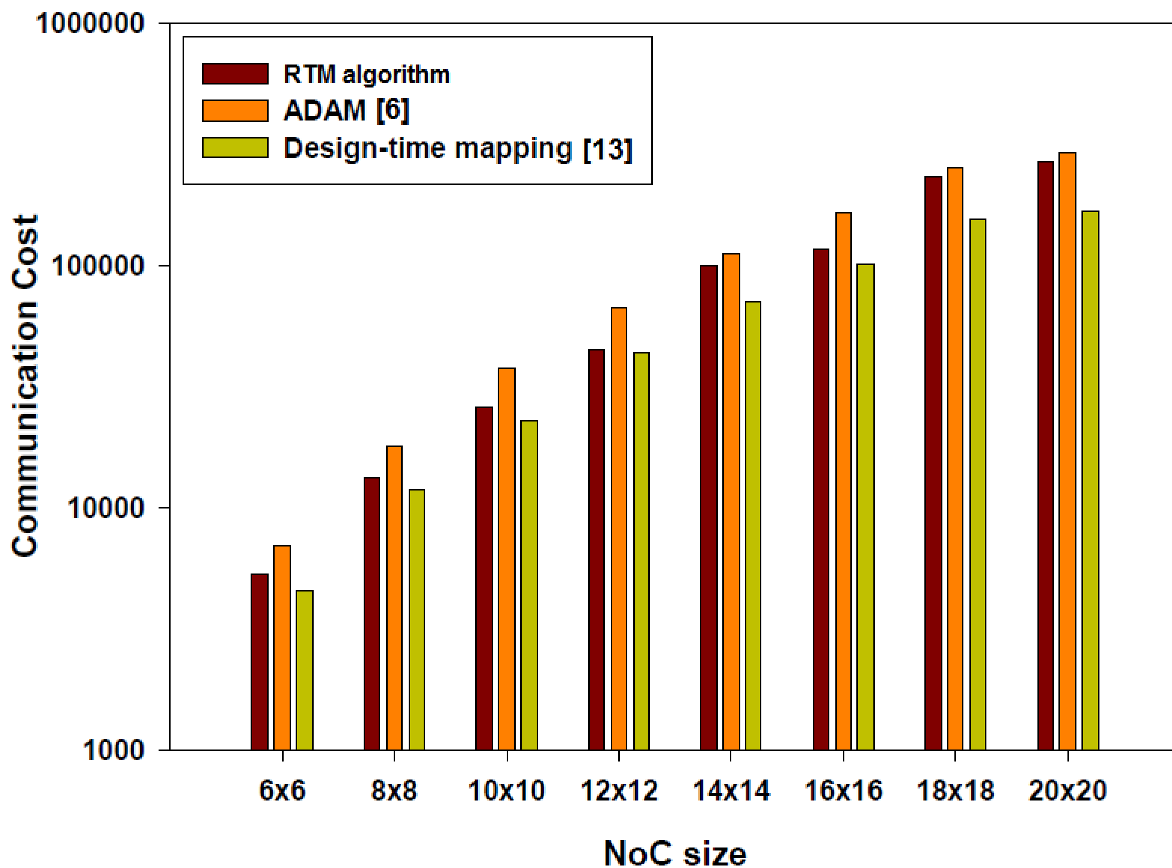


Figure 4.4: Communication Cost comparison in homogeneous platforms of various sizes.

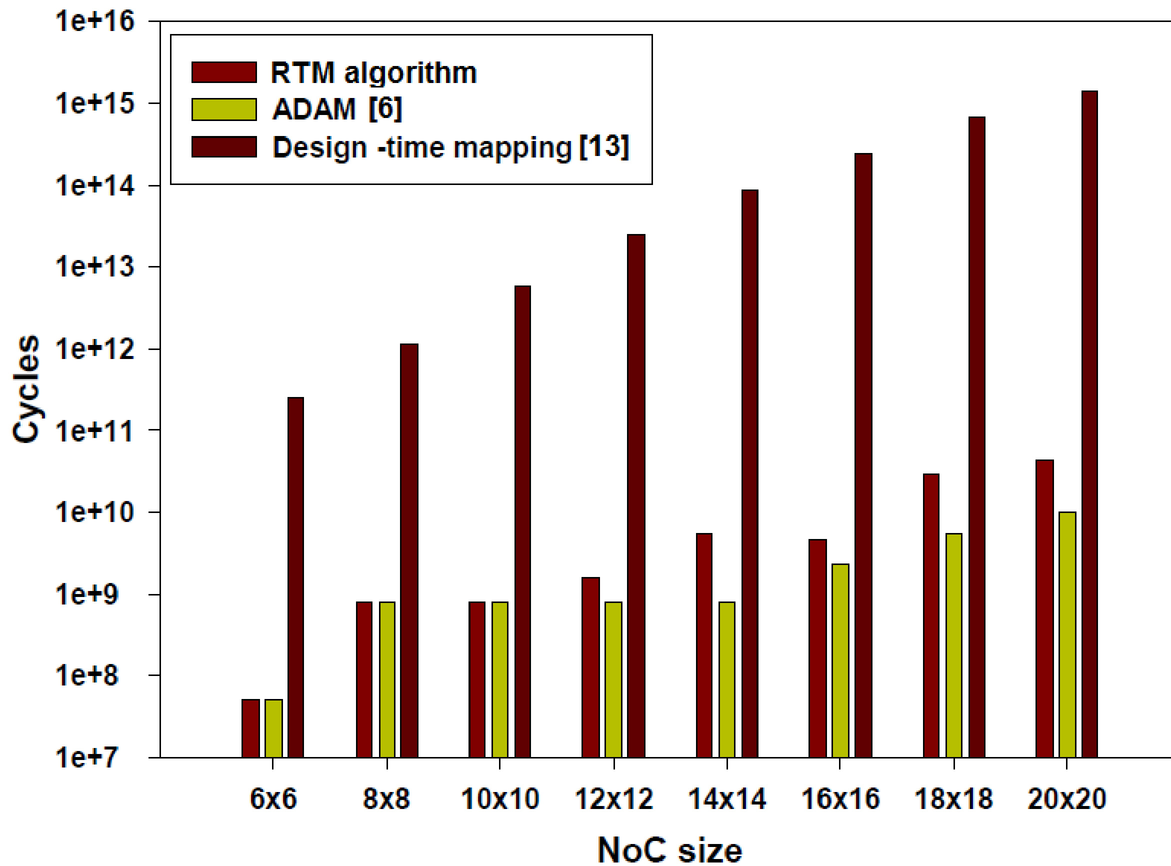


Figure 4.5: Mapping computational effort in homogeneous platforms of various sizes.

In figure 4.4, the algorithms are compared on terms of communication cost of the resulting mapping. The input is a single application, and the communication cost of the mapping is the one resulting from equation (2), that is, the one used for the swapping procedure. The algorithm is performed on various NoC sizes and proportional input task graphs. On figure 4.5, the computational effort for the three algorithms is shown, for the same applications in the same NoCs.

As expected, the best result is taken from the design-time mapping algorithm for every NoC size, due to its exhaustive search on the NoC. However, its execution time is huge compared to the other two algorithms and in addition it suffers from the constraints of design-time mapping. The RTM algorithm on the other hand results on average 22% more communication cost than the design-time algorithm, with as much as 6 orders of magnitude less computational effort, plus it is executed in run-time. As opposed to the ADAM algorithm, the RTM algorithm achieves up to 23% less cost with average 10% more computational effort. The extra computational effort on the RTM algorithm is due to the swapping procedure, but on most long-running applications, it is preferred to waste a few more cycles on mapping, rather than having more communication cost for the whole execution time of the application.

4.3.2. Application Benchmarks

In addition with the random application from tgff, the algorithm has also been tested on real application task graphs. These are:

- MPEG-4 (12 nodes)
- Multi-Window Display (MWD) (12 nodes)
- Picture-In-Picture (PIP) (8 nodes)
- MultiMedia System (MMS) (25 nodes)
- Digitale Radio Mondiale (DRM) (10 nodes)

The RTM algorithm is again compared with the ADAM and the exhaustive design-time algorithm in terms of communication cost. The computation effort isn't worth mentioning on these applications because of the low number of nodes results in menial differences between the 3 algorithms. The y-axis on the communication cost chart is again on logarithmic scale and the NoC size used on every application is noted on the chart.

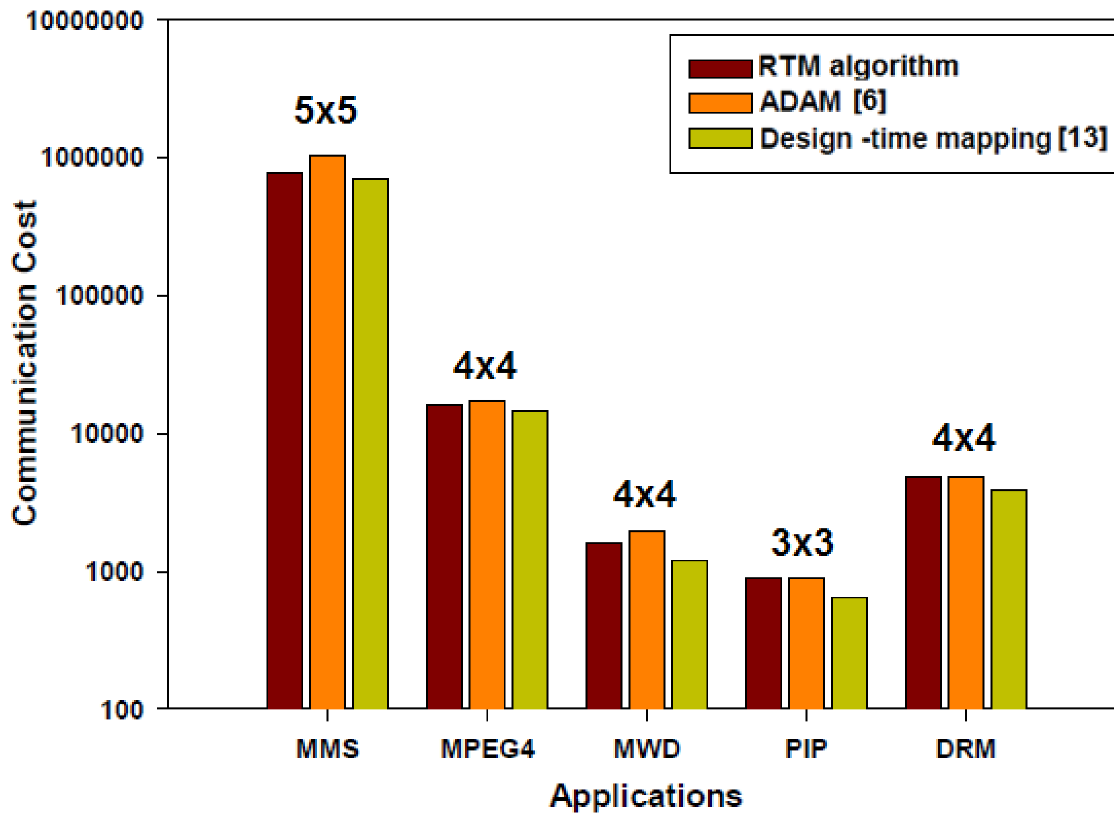


Figure 4.6: Mapping computational effort in homogeneous platforms for application benchmarks.

In figure 4.6, we can see that the results for the benchmarks agree with those of the tgff applications. The RTM algorithm has less communication cost than the ADAM algorithm and is close to the optimal solution produced by the exhaustive design-time algorithm. The difference of the RTM algorithm and the other two is presented in the following table :

Benchmark	**Improvement over the ADAM algorithm**	**Extra cost to the design-time mapping**
MMS	25%	11%
MPEG4	7.5%	7.9%
MWD	17%	35%
PIP	0%	40%
DRM	0%	25.6%

Table 4.1: Comparison of the RTM algorithm with the ADAM and design-time mapping algorithms

4.4. Experimental Results of the RTM algorithm for heterogeneous platforms

Here we present the experimental results of the algorithm on heterogeneous platforms. The input task graphs are once again generated with the tgff tool. In addition, the behavior of the algorithm is examined for scenarios of multiple applications whose requests arrive over time, in order to determine the differences of the mapping for different values of the Matching Factor.

4.4.1. TGFF generated applications

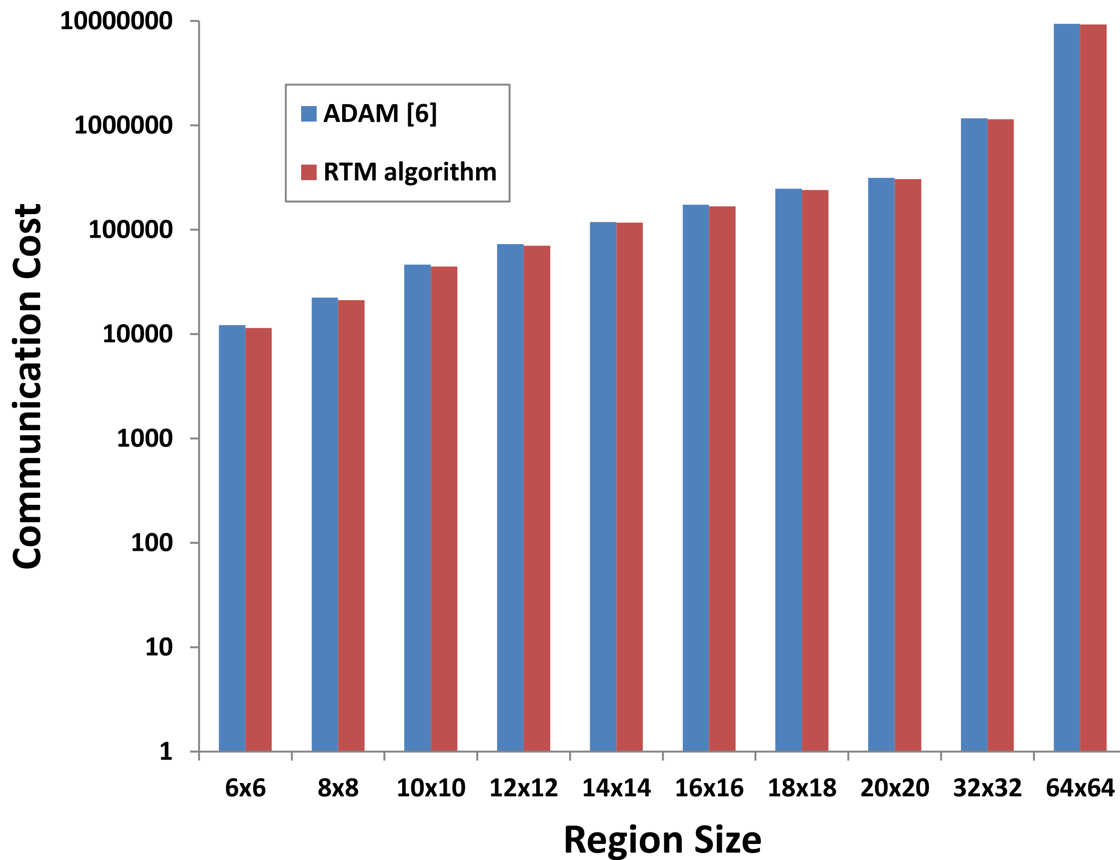


Figure 4.7: Communication Cost comparison in heterogeneous platforms of various sizes.

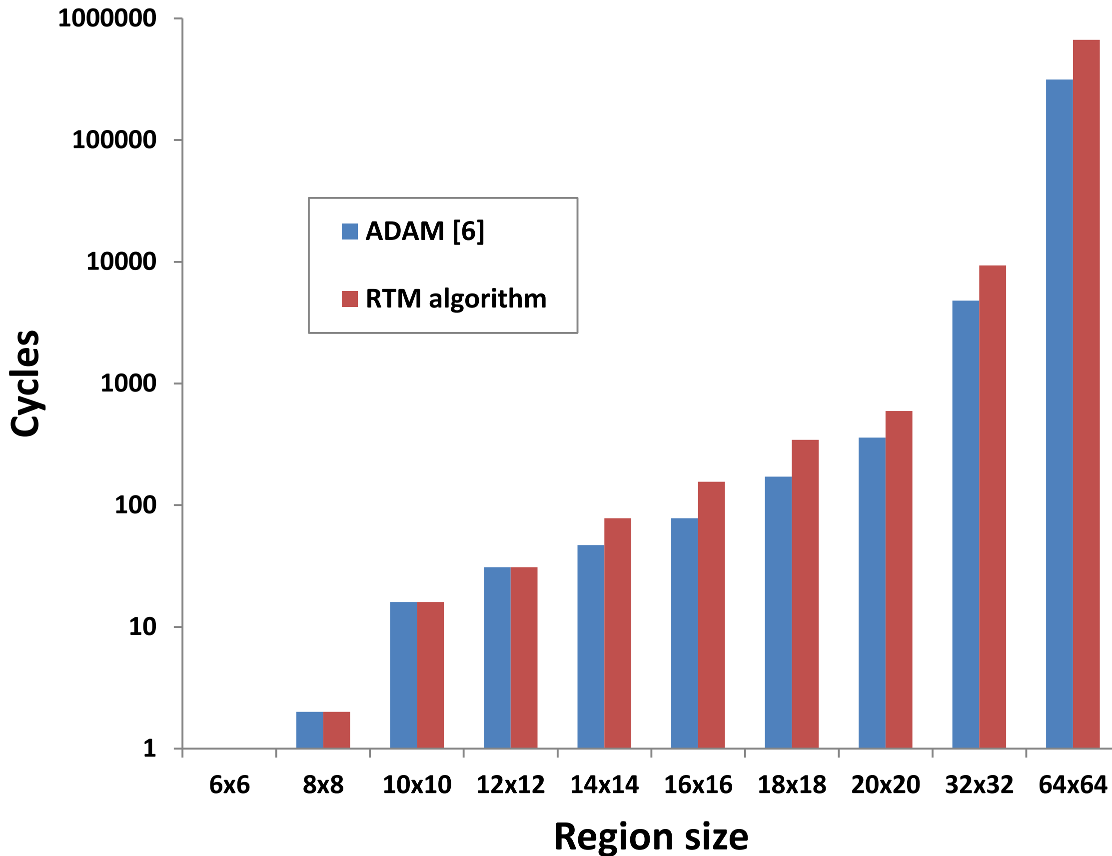


Figure 4.8: Mapping computational effort in heterogeneous platforms of various sizes.

The RTM algorithm is now compared to the ADAM algorithm. Since the platform in question is a heterogeneous NoC, there is a great diversity in the forms it can take, and it would be hard to try and examine all cases. Thus, we prefer to examine a common case and for that the regions used for the mapping have 3 different types of Processing Elements, in equal numbers. In figure 4.7 the communication cost is shown for the mapping of a single application on a region of various sizes, and also in figure 4.8 the computational effort needed for that mapping is depicted (including the region selection step that is performed on the System-Wide Controller). The size of the application is proportionate to the size of the region.

The RTM algorithm achieves up to 10% less communication cost than the ADAM algorithm, but needs on average 45% more time. This may be a big difference, but the mapping is computed in milliseconds, and thus the difference in the communication cost will probably be much more important and rewarding.

4.4.2. Utilization Scenarios

The results presented until now were about the mapping of a single application. These results may be representative for the mapping itself, but do not show the whole platform's behavior. That is best shown via the *simulation* of scenarios of multiple applications that arrive over time on a real platform. The scenarios have random applications arriving in random time intervals between them, considering that each task of each application is run for a certain amount of time and then frees its tile. These scenarios are mapped on a 6x6 heterogeneous NoC using the RTM algorithm for different MF values and the ADAM algorithm [6]. The goal of the utilization scenarios is to test which algorithm best utilizes the available resources of the system. The communication cost of each application after the mapping procedure has been presented in section 4.3.2. The platform used for the experiments is the one presented in section 4.1

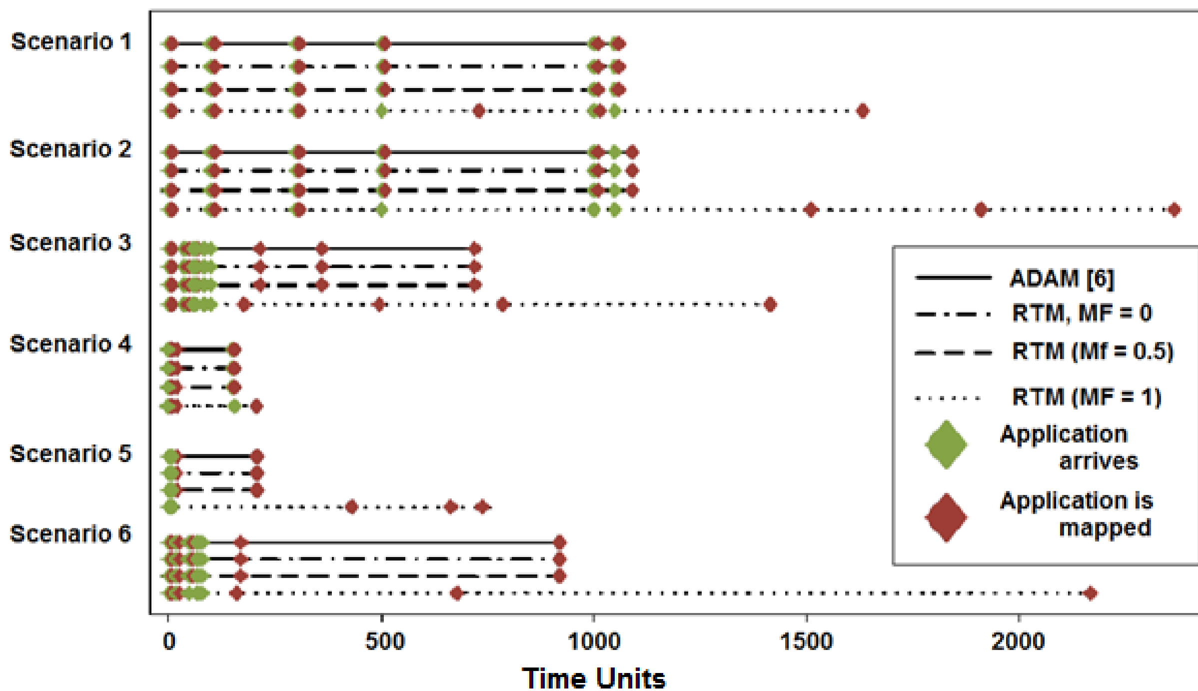


Figure 4.9: Run-time mapping scenarios on a heterogeneous NoC

Figure 4.9 depicts all the implemented scenarios. The green diamond represents the arrival time of an application while the red one represents the time that the mapping result was decided. The picture shows that both the RTM algorithm (with MF = 0 and MF = 0.5) has the same run-time behavior with the ADAM approach, and map most application at the moment they arrive, or close to that moment. The RTM algorithm with MF = 1 however, has a different behavior because under the MF = 1 restriction a task can be mapped only on a core that has the same Processing Element type with the task. In this case, the algorithm has to wait for the

desired cores to be freed from their previous applications, contrary to the other MF values or the ADAM algorithm that can change the types every task prefers, which results in mapping on other free tiles.

Even though it maps the applications later compared to the other algorithms, the RTM algorithm with MF = 1 has the best task to core mapping decision, resulting to best utilization of the platform's resources as depicted in Table 4.2. As *utilization* we consider the percentage of tasks that get mapped on a tile of their most preferred Processing Element type. Table 4.1 shows that with MF = 1, we can have 100% utilization of platform resources at run-time with a penalty cost at performance (*which could not even be correct since we have made the assumption that execution times are the same on every PE type*), but gaining greatly in energy consumption from the execution of the tasks. If the application needs are not so strict we can chose different values for the matching factor, thus relaxing the strictness of the matching.

	ADAM	RTM MF = 0	RTM MF = 0.5	RTM MF = 1
Scenario1	91%	92%	92%	100%
Scenario2	87%	88%	88%	100%
Scenario3	88%	87%	87%	100%
Scenario4	84%	86%	86%	100%
Scenario5	79%	78%	78%	100%
Scenario6	88%	87%	87%	100%

Table 4.2: Utilization percentages for the scenarios

4.5. Experimental results' conclusions

The results show that the RTM algorithm provides lower communication cost on the resulting mapping on almost all cases with some extra computational effort. This extra computational effort though is trivial in compared to the benefits of the lower cost, which, in the long run, reduces the total energy consumption of the chip. Furthermore, the novel idea of the Matching Factor can help in further increasing the energy efficiency of the platform. Following is the Table 2.1 filled with the RTM algorithm:

	Centralized/ Distributed	Homogeneous or Heterogeneous system	Implements: RT mapping and/or Task migration	Implementation difficulty	Testing Platform	Experimentation on :	Flexibility with various size NoCs and/or applications	QoS taken into consideration	Application profiling	Minimization of :
[6]:ADAM Run-time Agent-based Distributed Application Mapping for on-chip Communication	Mainly distributed with centralized elements (global agent)	Heterogeneous	Both	High	Undefined NoC's of various sizes	A robot application, multi-media applications and task graphs	High flexibility, thanks to clustering	Yes	Yes, tasks are classified by type.	Energy consumption
[7]:Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles	Centralized	Heterogeneous	Both	Medium	StrongARM processor of a PDA connected to an FPGA containing a 3x3 NoC of the PE's	Task graph with random application load and random platform load.	Bottleneck problem on large NoC's. Flexible with applications thanks to RH add-ons	Yes, task load specification function takes under consideration the user requirements	Yes, requested resource load and weights are calculated for each task	Internal fragmentation on reconfigurable tiles
[8]:Incremental run-time application mapping for homogeneous NoCs with multiple voltage levels	Centralized	Homogeneous	RT mapping	Low	6x6 NoC of AMD ElanSC520, AMD K6-2E and one MicroBlaze core	Synthetic Benchmarks	Bottleneck issues on large NoC's, good scaling with Application size	Yes, some tasks are considered critical and have tighter deadlines	Yes, critical tasks exist	Communication energy consumption
[9]:Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip (MPSOC)	Not specified	Heterogeneous	RT mapping	Medium	Hypothetical NoC consisting of ARM and Montium tiles.	HIPERLAN/2 receiver	Scales well with NoC size, but not with Application size (many iterations)	Yes	Yes, on design time	Energy consumption
RTM Algorithm	Distributed	**Both**	RT mapping	Medium	Nostrum NoC of Leon3 processor	Tgff applications and real applications benchmarks	High flexibility being distributed	Yes	Yes, tasks have preferences to PE types	Energy consumption and communication cost or maximization of utilization

Table 4.3: Comparison between state-of-the-art algorithms

Chapter 5:
Conclusions and Future work

5.1. Summary

In the current thesis, we studied the problem of run-time mapping of applications on a Network-on-Chip Multi-Processor System-on-Chip. After reviewing the principles of NoC architecture and related work, we address the problems of mapping on homogeneous and heterogeneous NoCs separately. For each of these problems a distributed run-time mapping algorithm has been developed, aiming to reduce the energy consumption and the communication costs from the execution of any application. Especially for heterogeneous systems, the concept of the Matching Factor is introduced, aiming in adding more flexibility and customization on the mapping procedure.

The developed RTM algorithm is compared with a state-of-the-art distributed run-time algorithm, both in homogeneous and heterogeneous platforms, showing 23% and 10% respectively lower communication cost in the resulting mapping, while having a small increase in the computational effort required. Also, for the heterogeneous platforms, the use of the correct MF value achieved on average 14% best utilization of the system's resources.

5.2. Future Work

The developed RTM algorithm could be improved and enriched with additional functionalities.

5.2.1 Task Migration

A very helpful addition would be the implementation of a task migration mechanism like the one presented in [7]. Task migration is the ability to re-allocate a task to a different tile after its initial mapping and during its execution (fig.5.1).

With this ability in hand, the system could potentially alter the mapping even while the application is being executed, in order to achieve either less energy consumption, or to better accommodate new applications that otherwise would not get mapped.

Furthermore, the task migration mechanism is needed in case the user requirements for an application change while it is executed, for example when switching to a different resolution in a video application.

Especially in the RTM algorithm on heterogeneous platforms task migration will prove very useful with MF = 100%. An example of its use is depicted in figure 5.2. This is the same example used in section 3.3.3. This time however, no region reshaping is needed, as the 3 tasks that are executed in region 4, are migrated in region 3, and the application can be mapped at once.

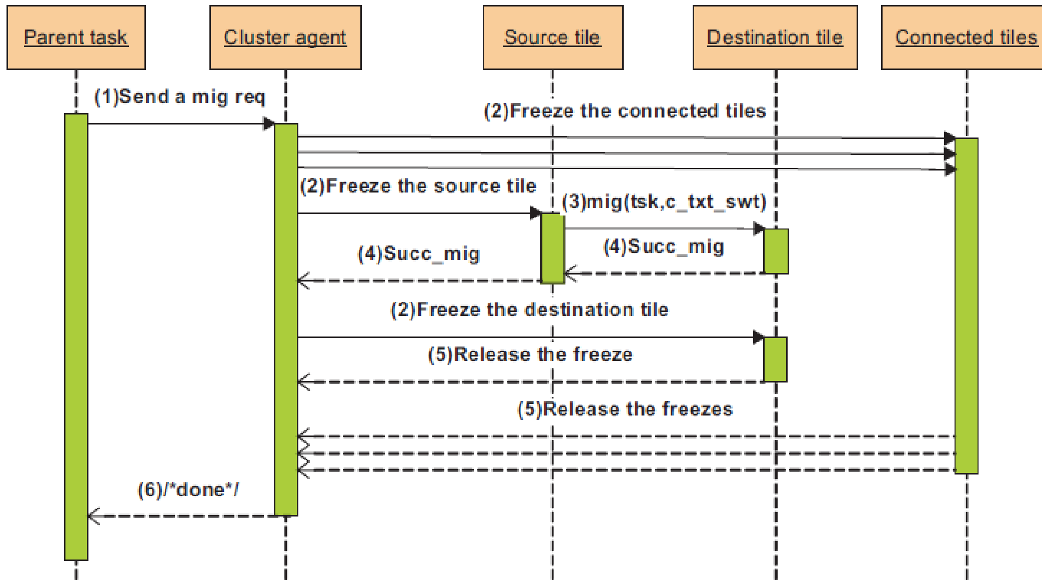


Figure 5.1: Task migration from [6]

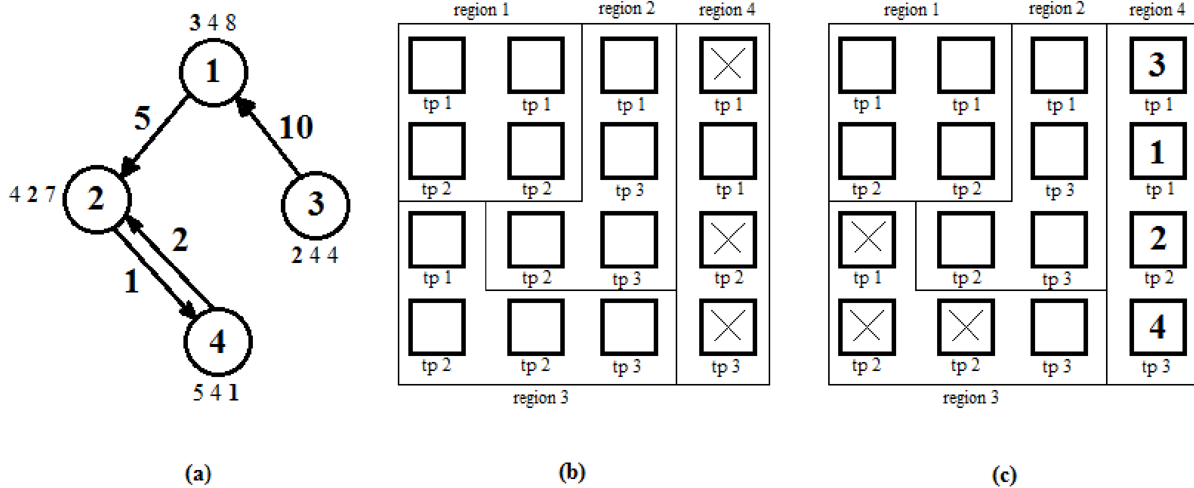


Figure 5.2: Example of the use of task migration
(a) Application task graph, (b) State of the NoC before the mapping, (c) The NoC after the mapping

5.2.2. Multitasking on the cores, Spatial and Temporal mapping

Another idea for future implementation would be the ability of scheduling on each tile, similar to the multitasking capability in modern single-core operational systems. This way, more

than one tasks could be mapped to every core and be executed in parallel, ending the problem of waiting for any other application to finish.

In order to achieve that, a dynamic two-level scheduling is needed, like the one presented in [14] (fig 5.3). Assuming a mechanism like that, the System-Wide Controller and the Regional Controller would continue mapping the tasks as it is, with the only difference being that more than one tasks would be able to get mapped on any core, even if they belong to different applications.

Then, a piece of code on each core, called the *local scheduler* would be in charge of deciding which task executes on the core, much like a multitasking scheduler of a common Operational System.

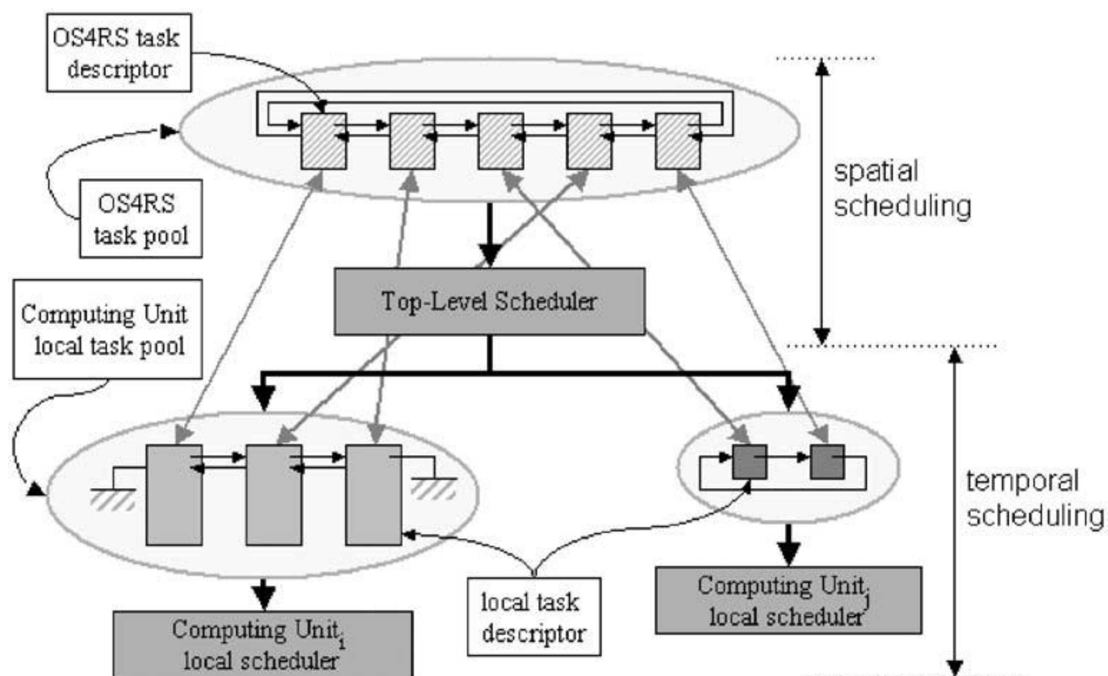


Figure 5.3: Two-level scheduling mechanism from [14]

5.2.3. High-level NoC control mechanisms for run time mapping

Last but not least, another addition would be the use of more hierarchy levels between the controllers on very large NoCs, i.e. the use of regions within regions, for better distribution of the algorithms computational effort. That would help on large NoCs, where possibly a region's Regional Controller would encounter the same problems as a centralized controller in a smaller platform.

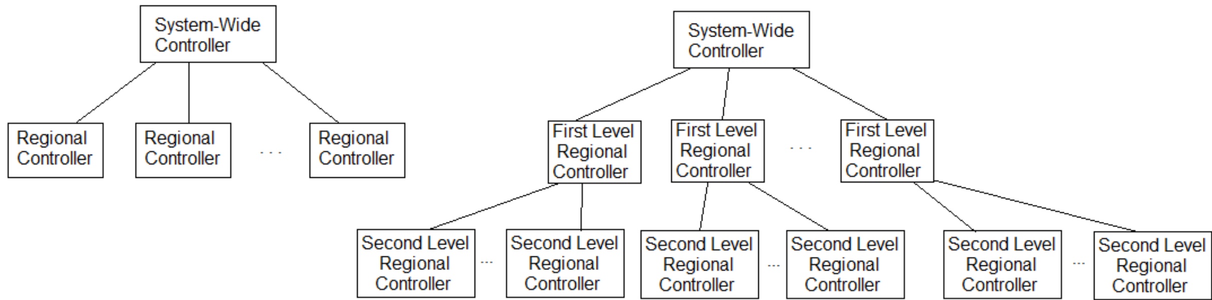


Figure 5.4: One-level and two-level hierarchical controllers

The RTM algorithm as it is, uses one-level of hierarchy on the controllers, utilizing the System-Wide Controller that is responsible for the Regional Controllers. Adding on more level of hierarchy would result in a hierarchy pyramid as the one depicted on figure 5.4. In this control scheme, the System-Wide Controller is responsible for the first level of Regional Controllers only, and these controllers are in turn responsible for the second level of Regional Controllers. In the same manner, as many levels as necessary can be added, in order to achieve the desired level of distribution on the NoC.

References

[1]: Tobias Bjerregaard and Shankar Mahadevan: *A Survey of Research and Practices of Network-on-Chip*. ACM Computing Survey Vol. 38, 2006

[2]: Axel Jantsch et al.: *Memory Architecture and Management in a NoC Platform*, In Axel Jantsch and Dimitrios Soudris, editors, *Scalable Multi-core Architectures: Design Methodologies and Tools*. Springer, 2011

[3]: Théodore Marescaux: *Mapping and Management of Communication Services on MP-SoC platforms*. Phd Thesis, IMEC, 2007

[4]: Ville Rantala, Teijo Lehtonen, Juha Plosila: *Network on Chip Routing Algoriths.* TUCS Technical Report, 2006

[5]: Wayne Wolf: *Computers as components*: *principles of embedded computing system design*, Morgan Kaufmann, 2001

[6]: Mohammad Abdullah Al Faruque et al.: *Adam: run-time agent-based distributed application mapping for on-chip communication.* DAC 2008

[7]: V. Nollet et al.: *Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles,* in Proc. of DATE. IEEE Computer Society, 2005

[8]: Chen-Ling Chou, Radu Marculescu: *Incremental Run-time Application Mapping for Homogeneous NoCs with Multiple Voltage Levels,* in Proc. of CODES+ISSS. ACM, 2007

[9]: Philip K.F. Hölzenspies et al.: *Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip (MPSoC).* 2008 DATE

[10]: Luca Benini, Giovanni De Micheli: *Networks on Chips: Technology and Tools*, Morgan Kaufmann, 2006

[11]: Jingcao Hu, Radu Marculescu: *Energy- and Performance- Aware Mapping for Regular NoC Architectures.* IEEE Trans. On CAD of Integrated Circuits and Systems

[12]: R.P. Dick et al.: *Tgff: task graphs for free*, in CODES'98, 1998

[13]: Iraklis Anagnostopoulos, Alexandros Bartzas, Kostas Siozios, Dimitrios Soudris: *High-level customization methodology for application-specific NoC architectures,* 2011

[14]: T. Marescaux et al.: *Run-time support for heterogeneous multitasking on reconfigurable SoCs.* Elsevier 2004