RUN-TIME MAPPING TECHNIQUES FOR NOC-BASED HETEROGENEOUS MPSOC PLATFORMS

AMIT KUMAR SINGH

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2013

Acknowledgments

I would like to take this opportunity to thank many people who directly and indirectly supported me all through the last four years. Without their help and support, this thesis would not have reached its current form.

First of all I would like to express my heartfelt gratitude to my supervisor Prof. Thambipillai Srikanthan for his invaluable guidance, support and suggestions. Despite being entrusted with many responsibilities, he always found time for discussion with me. His patience and understanding in this regard is truly admirable.

I would like to thank Mr. Ewerson Carballo for providing the basic simulation environment that we extended for verifying our initial ideas and to Dr. Wu Jigang, Alok, Samarth and Dr. Akash Kumar for our discussions on ideas and implementation issues. I would especially like to thank Dr. Akash Kumar who always took time to understand the problems that I faced during my Ph.D. and provided sound suggestions.

The administrative support rendered by Ms. Nah, Merilyn and Jeremiah was the best I could have asked for. They always helped me out whenever I needed any equipment or administrative work to be sorted out. I want to thank them for all the help they have provided me at CHiPES.

This acknowledgement is incomplete without mentioning my dear friends with whom I have shared many wonderful moments. They always cheered me up with their jokes and antics. I want to thank Alok (Bihari), Shantanu (Lambu), Ashish (Dada), Bharath (Motu), Abhijit (Taklu Bhai), Sachan (Baba), Suvu and Manish for all the great times.

Last but not the least; I would like to thank my family for their constant love and encouragement. I would especially like to thank my wife Arpita, who endured all the late evenings, early mornings, week-ends and vacations that were consumed in producing this thesis. I would not have reached this point without her loving support. I feel truly blessed to have such a strong support system around myself.

Contents

List of Figures

List of Tables

List of Abbreviations

ACC	Accelerator
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
BN	Best Neighbor
CCR	Computation-Communication Ratio
CPBN	Communication-aware Packing-based Best Neighbor
CPNN	Communication-aware Packing-based Nearest Neighbor
DSE	Design Space Exploration
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
HDSE	Heuristic Design Space Exploration
HetEDSE	Heterogeneous Exhaustive Design Space Exploration
HetPDSE	Heterogeneous Pruning-based Design Space Exploration
HomEDSE	Homogeneous Exhaustive Design Space Exploration
HomPDSE	Homogeneous Pruning-based Design Space Exploration
HRM	Hybrid Run-time Mapping
IP	Intellectual Property
ISA	Instruction Set Architecture
ISM	Ideal Static Mapping
ISP	Instruction Set Processor
ITRS	International Technology Roadmap for Semiconductors

MIPS	Million Instructions Per Second
MPSoC	Multi-Processor System-on-Chip
NN	Nearest Neighbor
NoC	Network-on-Chip
NP-hard	Non-deterministic Polynomial-time hard
NRE	Non Recurring Engineering
PBN	Packing-based Best Neighbor
PE	Processing Element
PHeterog	Preprocessing-based Heterogeneous
PHomog	Preprocessing-based Homogeneous
PNN	Packing-based Nearest Neighbor
PTBN	Packing-based Time-bounded Best Neighbor
RA	Reconfigurable Area
RH	Reconfigurable Hardware
RISC	Reduced Instruction Set Computing
RTM	Run-time Manager
SDFG	Synchronous Dataflow Graph
SoC	System-on-Chip
VLSI	Very Large Scale Integration

Abstract

The reliance on Multi-Processor Systems-on-Chip (MPSoCs) to satisfy the high performance requirement of complex embedded software applications is increasing. The Networks-on-Chip (NoCs) based interconnection infrastructure is fast becoming a preferred approach to facilitate communication among the processing elements (PEs) of MPSoCs. The heterogeneity of MPSoCs is also increasing by employing different types of PEs in order to meet the functional and non-functional requirements. This necessitates the need to realize efficient run-time mapping techniques for such heterogeneous computing platforms.

In this thesis, a number of efficient techniques have been proposed to realize run-time mapping algorithms for heterogeneous MPSoC platforms. MPSoC with single-task supported PEs, each of which consisting of a general purpose processor or reconfigurable hardware is considered first. A new packing strategy to map the various tasks of an application in close proximity has been proposed to reduce the communication overhead. The proposed strategy was further extended to devise a time-bounded method to minimize the overall execution time of the mapping process. Performance evaluations based on 20 random applications show that the proposed techniques outperform the existing techniques by up to 22%.

Subsequently, the proposed mapping process was extended to support an MPSoC platform in which each PE is capable of supporting multiple tasks. The extended techniques facilitate in the mapping of a group of communicating tasks on the same PE, thereby resulting in a further reduction in the communication overhead. The extended time-bounded method reduces the time required to identify the best mapping configuration. Moreover, the overall communication overhead is also reduced, resulting in improved performance. On average, channel load and total energy consumption is reduced by 10% and 46% respectively.

The run-time mapping techniques were further enhanced by taking into account both the computation and communication costs so as to optimize the overall computation efficiency. They rely on the systematic elimination of the longest communication path at a time until the computation load on any PE impedes on the overall performance. The proposed techniques were tested using multiple scenarios of an MPEG-4 application to demonstrate that the total execution time and energy consumption can be reduced by 33% and 39% respectively when compared to an approach that rely solely on the communication-aware strategy.

A hybrid strategy has also been proposed to further accelerate the run-time mapping process when the applications to be supported on a platform are known at design-time. It relies on the efficient design-time analysis to generate light-weight run-time mapping heuristics, which aid the communication and computation aware run-time mapping process. Experiments based on models of real-life multimedia applications show that the proposed analysis strategy is faster by 83% and run-time mapping is accelerated by 93% when compared to state-of-the-art analysis and on-the-fly mapping approaches, respectively. Finally, the proposed run-time mapping process relies on an efficient computation and communication aware mapping strategy, which is complemented by light-weight heuristics to implement embedded computing applications that demand high performance.

Chapter 1

Introduction

1.1 The Multi-Processor System-on-Chip Revolution

When looking closely at our modern electronic systems, it is clear that we have entered the Multi-Processor System-on-Chip (MPSoC) era. In essence, this era is initiated by the need to deal with complex applications.

In 1965, Moore's law predicted that the number of transistors in the same chip area will grow exponentially [1]. The growing trend is shown in Figure 1.1. With the growing trend, digital electronic devices capabilities such as processing speed, memory capacity, even the number of pixels in digital cameras have increased with roughly the same exponential rate. This has required simultaneous attention in two directions. On one hand, hardware designers need to provide bigger, better and faster means of processing, and on the other hand, the application developers have to maximize the utilization of the processing power. The rising high level of integration enables implementation of multiple processors within a single chip towards the development of MPSoC [3].

Intel's first released processor 4004 in 1971 had approximately 2,300 transistors (Figure 1.1). This processor operated at a speed of 400 KHz. In contrast, a modern single processor chip (for example, Intel Pentium 4) has more than a billion of transistors operating at more than 3 GHz. Going forward, as the maximum operational frequency of a processor has hit the roof due to power dissipation and radio frequency effects, chip

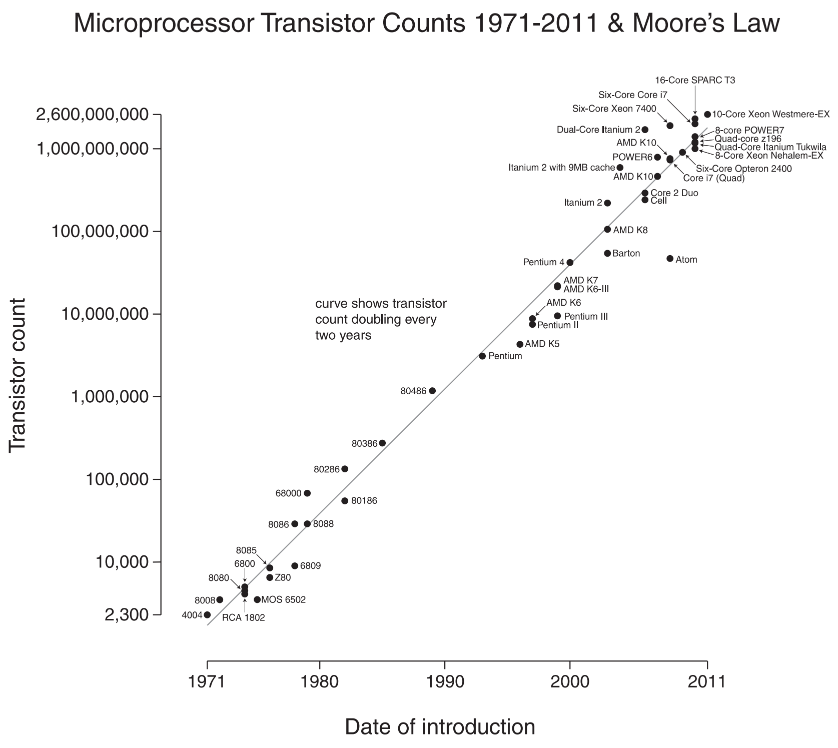


Figure 1.1: Continuation of Moore's Law [1]

manufacturers are forced to limit the maximum frequency of the processor and shifting towards designing chips with multiple processors operating at lower frequencies. In Figure 1.1, it is interesting to observe the introduction of dual core processor chips from 2005 onwards. This indicates the beginning of MPSoC era.

Moreover, the rising complexity of modern real-life applications cannot be handled by simply increasing the frequency of a single-core processor. Instead, it requires multiple processors which can cohesively communicate and provide increased parallelism. The underlying concept is to consider applications as conglomeration of many parallelized small tasks which can be efficiently distributed on multiple processors in order to execute them in parallel and thereby meeting the increased performance demand of complex applications.

1.2 Motivation

It is a well known fact that customizing a single processor for the application can improve performance. However, the performance demands of modern complex embedded applications have increased substantially which can only be satisfied using multiple processors providing increased parallelism. The communication requirement of large number of processors can be satisfied by efficient Networks-on-Chip (NoCs). The processors need to be of different types for executing different tasks efficiently in order to achieve better performance. With the presence of reconfigurable hardware blocks in heterogeneous MPSoCs, the hardware blocks can be configured at run-time according to the functionality needed by compute intensive tasks as it provides flexibility at similar level to that of the general purpose processors. The acceleration provided by the hardware can be used to satisfy the imposed performance demands. Similarly, other types of processors can be configured to exploit them at their maximum capacity. Thus, heterogeneous MPSoCs will be required as the performance demands increase.

Modern embedded systems (e.g., smart phones, PDAs, tablet PCs) employ MPSoCs in order to support multiple applications concurrently. For example, a smart phone might be used to view an image using a JPEG decoder application over the internet and at the same time to listen to music using an MP3 decoder application. The components (tasks and their connections) of applications need to be mapped and scheduled on the MPSoC resources efficiently in order to satisfy performance constraints for each application. Mapping and scheduling problem is similar to *Quadratic Assignment Problem*, a well known NP-hard problem [4]. Therefore, finding optimal solution satisfying all the given constraints is very difficult and time consuming. For example, exploring all the tasks to resources combinations exhaustively and then choosing the optimal combination may take days or weeks for a large number of tasks and processors. Thus, heuristics based on the application domain knowledge need to be employed to find a nearly optimal solution.

Mapping applications on an MPSoC platform can be accomplished at either design-time or run-time. The design-time mapping techniques are suitable only for static workload scenarios and thus are unable to handle dynamism in applications incurred at run-time (e.g., multimedia and networking applications). Since applications are often added to the platform at run-time (for example, downloading a Java application in a mobile-phone at run-time), workload variation takes place. We witness the need of run-time mapping techniques to handle such dynamic workloads. The run-time mapping techniques face the challenge to map new applications on the platform resources with accurate knowledge of resource occupancy in order to satisfy their performance requirements.

At run-time, new applications can be mapped with or without previously analyzed results based on different kind of scenarios. When the applications to be supported on a platform are not known at design-time, they need to be mapped without any previous analysis. This requires efficient heuristics to be defined to assign new arriving tasks on the platform resources. Such heuristics perform all the processing at run-time. They cannot guarantee for schedulability, i.e., for strict timing deadlines due to lack of any prior analysis and limited compute power at run-time. However, these heuristics are platform independent since they do not use any platform specific analysis results computed in advance.

The applications to be supported on a platform should be known at design-time in order to map them using previously analyzed results. In such cases, light-weight heuristics are required to select the most efficient mappings for each application from the design-time (offline) analyzed mappings stored on the system. The selection should be subject to available system resources and desired performance. The mappings should contain schedules and allocations. The selected mappings can be used to configure the platform.

The design-time analysis needs to perform all the compute intensive processing. This facilitates for light-weight run-time platform manager that can configure the applications efficiently. Design-time analysis to explore mappings, i.e. tasks to resources allocations exhaustively is not feasible within a limited time for large application and platform size. Therefore, faster analysis strategies exploring all the efficient mappings need to be developed. These strategies need to consider platform specifications for performing exploration so the analysis results will not be applicable to all the platforms.

1.3 Aim of the Research

The main aim of this research is to develop efficient techniques and methodologies for run-time mapping of embedded applications on heterogeneous MPSoC platforms in order to maximize performance. The proposed techniques must consider NoC-based platforms as NoC is highly scalable which can cater for the larger platforms expected in future. The techniques need to be scalable for large problems as well.

1.4 Main Contributions of the Thesis

We provide an overview of the main contributions that have been made during the course of this research. The main contributions can be summarized as follows:

1. *Run-time mapping techniques for efficiently mapping applications on NoC-based heterogeneous MPSoCs containing single task supported processing elements.* We show that the existing mapping techniques may not lead to the best results. The proposed techniques map tasks of the applications on the MPSoC processing elements (PEs) in very systematic manner, leading to efficient mapping. These techniques show performance improvement when compared to existing techniques.

2. *Extending the run-time mapping techniques to heterogeneous MPSoCs containing multi-task supported PEs.* Supporting a single task on each PE is not a realistic scenario. Additionally, when multi-task supported PEs are considered, we cannot exploit possible advantages of the PEs by the techniques considering single task supported PEs. The extended techniques take advantage of the multi-task supported PEs and show performance improvements.

3. *Techniques for communication-aware run-time mapping of applications on MPSoCs.* State-of-the-art mapping techniques do not consider communication between tasks during mapping and thus are not able to exploit the multi-task supported PEs efficiently. The proposed techniques consider communication between tasks while performing mapping, resulting in reduced communication overhead when compared to state-of-the-art mapping alternatives. Reduced communication overhead leads to significant performance improvement.

4. *Techniques for computation and communication aware run-time mapping.* Computation and communication aware mapping is required when both computation and communication overhead are of significant importance. The proposed techniques perform preprocessing of the applications by considering computation and communication load balancing between tasks before application tasks are actually allocated to the platform resources. In preprocessing, communication bottlenecks are repeatedly removed till a processor becomes bottleneck. Thereafter, resource optimization is carried out to get the final preprocessed graph. We show that the proposed techniques outperform existing mapping techniques.

5. *Hybrid mapping strategy for accelerating run-time mapping.* The strategy uses design-time analysis results while performing run-time mapping. We show that the run-time mapping using previously analyzed results gets accelerated as the computation intensive processing is performed at design-time.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows:

- **Chapter 2:** This chapter provides a background review on need for MPSoC architectures in modern embedded systems, challenges and methodologies to design MPSoCs as well as techniques and methodologies to map applications on MPSoCs. Subsequently, the existing mapping strategies are analyzed to highlight their limitations and the most appropriate strategies are identified for further investigation.

- **Chapter 3:** We propose run-time mapping techniques to map the applications on MPSoCs containing single task supported PEs. The mapping techniques are based on a packing strategy that maps the tasks in close proximity in order to reduce communication overhead of communicating tasks.

- **Chapter 4:** We extend the run-time mapping techniques proposed in Chapter 3 so that more than one tasks can be supported on the PEs. The extended techniques take the advantage of the multi-task supported PEs and provide better performance.

- **Chapter 5:** In this chapter, we present communication-aware mapping techniques that consider communication between tasks during mapping in order to exploit the multi-task supported PEs efficiently. The presented techniques map maximum communicating task pairs on the same PE, resulting in reduced communication overhead.

- **Chapter 6:** In this chapter, we further enhance the mapping techniques to consider both computation and communication overhead. Unlike the techniques proposed in Chapter 5 that consider communication overhead only, the techniques proposed in this chapter perform efficient mapping by taking computation overhead, communication overhead and resource utilization into account. We show that the proposed techniques outperform the communication-aware techniques for the scenarios where both computation and communication overhead are significant.

- **Chapter 7:** In this chapter, we present a hybrid mapping strategy that performs compute intensive analysis at design-time and uses the analysis results at run-time for facilitating efficient mapping. We also present analysis strategies performing faster design space exploration and at the same time providing better design points when compared to existing analysis strategies. The analysis results are used at run-time by a proposed run-time mapping strategy in order to accelerate the mapping process.

- **Chapter 8:** We conclude the thesis and identify some future directions in this work.

1.6 List of Publications resulting from the Thesis

The work presented in this thesis has been communicated in international journals and conferences as follows:

International Refereed Journals

[J-1] A. K. Singh, T. Srikanthan, A. Kumar and W. Jigang, "Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms", Journal of Systems Architecture, Vol. 56, No. 7, July 2010, pp. 242-255.

[J-2] A. K. Singh, A. Kumar and T. Srikanthan, "Accelerating Throughput-aware Run-time Mapping for Heterogeneous MPSoCs", ACM Transactions on Design Automation of Electronic Systems. (Accepted for Publication)

[J-3] A. K. Singh, A. Kumar, W. Jigang and T. Srikanthan, "CADSE: Communication Aware Design Space Exploration for Efficient Run-time MPSoC Management", Frontiers of Computer Science. (Accepted for Publication)

International Refereed Conferences

[C-1] A. K. Singh, W. Jigang, A. Prakash and T. Srikanthan, "Efficient Heuristics for Minimizing Communication Overhead in NoC-based Heterogeneous MPSoC Platforms", IEEE/IFIP International Symposium on Rapid System Prototyping (RSP), June 2009, pp. 55-60.

[C-2] A. K. Singh, W. Jigang, A. Prakash and T. Srikanthan, "Mapping Algorithms for NoC-based Heterogeneous MPSoC Platforms", IEEE Euromicro Symposium on Digital Systems Design (DSD), August 2009, pp. 133-140.

[C-3] A. K. Singh, W. Jigang, A. Prakash, T. Srikanthan and D. Maskell, "Efficient Task Mapping in Multi-tasking Heterogeneous MPSoC Platforms", Asia-Pacific Embedded Systems Education and Research Conference (APESER), December 2009.

[C-4] A. K. Singh, W. Jigang, A. Kumar and T. Srikanthan, "Run-time Mapping of Multiple Communicating Tasks on MPSoC Platforms", International Conference on Computational Science (ICCS), June 2010, pp. 1013-1020.

[C-5] A. K. Singh, A. Kumar, T. Srikanthan and Y. Ha, "Mapping Real-life Applications on Run-time Reconfigurable NoC-based MPSoC on FPGA", IEEE International Conference on Field Programmable Technology (FPT), December 2010, pp. 365-368.

[C-6] S. Kaushik, A. K. Singh and T. Srikanthan, "Preprocessing-based Run-time Mapping of Applications on NoC-based MPSoCs", IEEE Annual Symposium on VLSI (ISVLSI), July 2011, pp. 337-338.

[C-7] A. K. Singh, A. Kumar and T. Srikanthan, "A Design Space Exploration Methodology for Application Specific MPSoC Design", IEEE Annual Symposium on VLSI (ISVLSI), July 2011, pp. 339-340.

[C-8] S. Kaushik, A. K. Singh and T. Srikanthan, "Computation and Communication Aware Run-time Mapping for NoC-based MPSoC Platforms", IEEE International SOC Conference (SOCC), September 2011, pp. 185-190.

[C-9] A. K. Singh, A. Kumar and T. Srikanthan, "A Hybrid Strategy for Mapping Multiple Throughput-constrained Applications on MPSoCs", IEEE/ACM International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES), October 2011, pp. 175-184.

[C-10] S. Kaushik, A. K. Singh, W. Jigang and T. Srikanthan, "Run-Time Computation and Communication Aware Mapping Heuristic for NoC-based Heterogeneous MPSoC Platforms", IEEE International Symposium on Parallel Architectures, Algorithms and Programming (PAAP), December 2011, pp. 203-207.

[C-11] A. K. Singh, A. Kumar, W. Jigang and T. Srikanthan, "Communication-aware Design Space Exploration for Efficient Run-time MPSoC Management", IEEE International Symposium on Parallel Architectures, Algorithms and Programming (PAAP), December 2011, pp. 72-76.

[C-12] A. K. Singh "Run-Time Mapping Techniques for NoC-based Heterogeneous MPSoC Platforms", EDAA/ACM SIGDA PhD Forum at the Design Automation and Test in Europe (DATE) Conference, March 2012.

Chapter 2

Literature Survey

Modern embedded systems (e.g., smart phones, PDAs, tablet PCs) employ Multi-Processor Systems-on-Chip (MPSoCs) in order to satisfy the ever-rising performance demands of modern complex embedded applications while also reducing power consumption. Therefore, MPSoC platforms consisting of several embedded processors are becoming ubiquitous in embedded processing [5]. These platforms can provide increased performance by executing parallel tasks of applications on different processors at the same time. Further, the processors operate at lower frequencies unlike at a very high frequency in single-core processor based systems and thus fulfilling the low power requirement. Intel reports that under-clocking a single core by 20 percent saves half of the power while sacrificing just 13 percent of the performance [6]. So, if the work is distributed on two processors running at 80 percent clock rate, we get 74 percent better performance for the same power. However, the heat is dissipated at two points rather than one.

The MPSoC platforms are bound to contain larger number of processing elements (PEs) as technology advances. The platforms can be homogeneous or heterogeneous depending upon the type of PEs present in the platform. Homogeneous platforms contain identical PEs making them very suitable for VLSI implementation. On the other hand, heterogeneous platforms contain different type of PEs in order to satisfy higher performance demands by exploiting distinct features of the PEs. The platform PEs call for a communication infrastructure to have proper communication amongst multiple PEs. Mapping of applications on the platform PEs by using efficient mapping techniques is very active topic of research and is being addressed by several research organizations [7] [8].

In this chapter, we discuss the trends in MPSoC in Section 2.1. Section 2.2 introduces some of the existing homogeneous and heterogeneous MPSoC platforms targeted for different domain of embedded applications. In the same section, we discuss available on-chip interconnects required to fulfill the communication needs of various platform PEs and the challenges involved in designing MPSoC platforms along with various existing design methodologies. Various mapping techniques proposed in the literature to map applications on MPSoC platforms have been discussed in Section 2.3. A comprehensive review of the mapping techniques is undertaken to highlight their limitations, which gave us the main motivation for this dissertation. The contents of this chapter are summarized in Section 2.4.

2.1 Trends in Multi-Processor System-on-Chip

This section describes the trends in the MPSoC as technology evolved and performance demand increased.

2.1.1 Number of Processing Cores

Continuing with the Moore's law, the number of transistors will grow exponentially, thereby the number of cores with roughly the same exponential rate. Moreover, International Technology Roadmap for Semiconductors (ITRS) predicts that this growing trend will continue as shown in Figure 2.1. Thus, as nanotechnology evolves, it will become feasible to integrate thousands of cores on the same chip as predicted by sources like Intel and Berkeley [9] [10]. The cores are envisioned as logic gates of 21^{st} century.

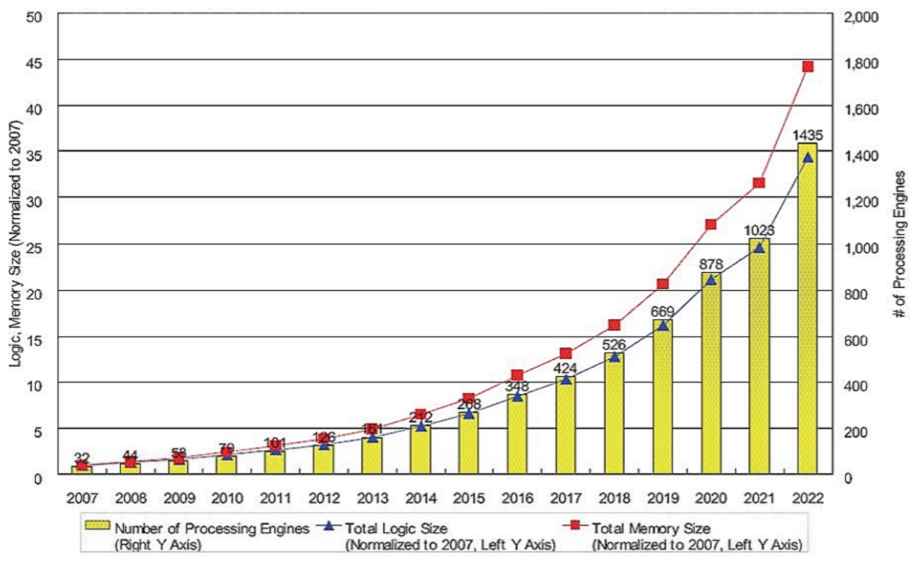


Figure 2.1: ITRS Roadmap showing growing number of processing cores (engines) [11]

Almost all computing vendors have announced chips with multiple processor cores. Moreover, the vendor road-maps assure for repeatedly doubling the number of cores per chip. These chips are to be used in future and are diversely called chip multiprocessors, multi-core chips, and many-core chips. The complete systems are usually referred to as MPSoCs.

2.1.2 Network-on-Chip for Scalability

The processors present in the MPSoCs call for a communication infrastructure to have proper communication amongst them. This communication infrastructure can be based on buses, point-to-point links or Networks-on-Chip (NoCs) [12]. In bus-based infrastructure, as the number of processors increases, the arbitration bottleneck increases due to need of increased number of bus masters. Additionally, the bus bandwidth gets shared by all the attached processors, making it non-scalable. In point-to-point links, with increased number of processors, longer wires are required, causing long delays in communication. Thus, this infrastructure too is non-scalable. However, in a NoC, the arbitration is distributed and only wire segments are required. Further, the bandwidth gets scaled with the network size, i.e. number of processors. Thus, NoC communication infrastructure is efficient and highly scalable [13] [14].

2.1.3 Heterogeneity in Processing Cores

Amdahl's law has been augmented in multi-core era to evaluate the true benefits of multi-core processing [15]. It states that the speedup of an application by MPSoC processing is limited by the time needed to execute the sequential portion of the application. Figure 2.2 shows the speed up obtained by using different number of processors at various levels of parallelization. It is clear that if 5% of the application cannot be parallelized (95% parallelized) then the maximum speedup that can be achieved is 20x even if larger number of processors is used.

The speedup can be increased by accelerated execution of the non-parallelized part (5%), i.e. by executing the sequential part in less time with the help of a processor that has better sequential performance. When heterogeneous MPSoCs are considered, the distinct features of different type of processors can be exploited by different portions of the application which might lead to increased speedup. Thus, heterogeneous MPSoCs have become formidable computing alternatives where applications witness large improvement over their homogeneous counterpart.

Further, heterogeneous MPSoCs may contain general purpose processor (GPP) for flexibility, custom accelerators for compute intensive tasks, reconfigurable hardware blocks for flexibility & compute intensive processing and specialized processors like digital signal processors (DSPs) for signal processing tasks, thereby providing flexibility, increased compute performance and reduced power consumption at the same time. Going forward, heterogeneity can be increased further to achieve high performance demands of complex applications.

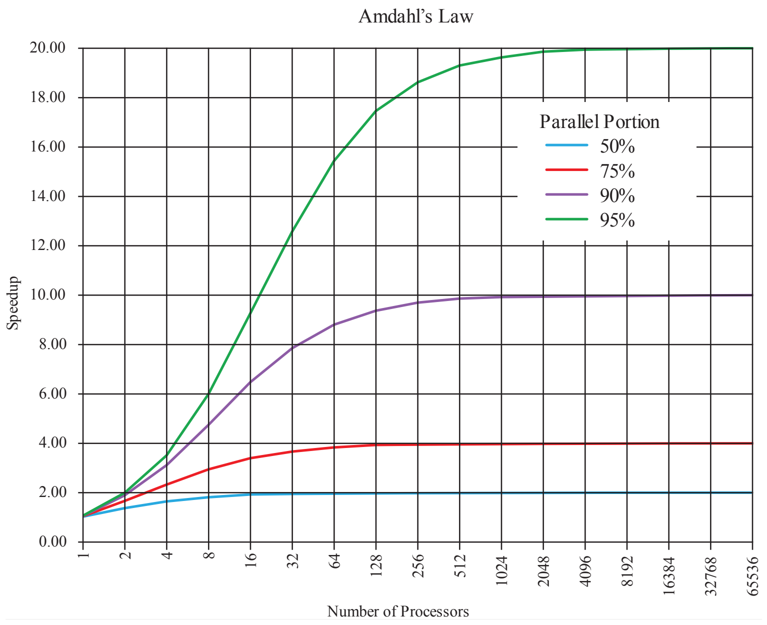


Figure 2.2: Amdahl's law indicating that speed up obtained using multiple processors is limited by the sequential part of the program [16]

2.2 Multi-Processor System-on-Chip Architectures

A Multi-Processor System-on-Chip (MPSoC) has two aspects - multiple processors and System-on-Chip. A multi-core architecture approach can be adopted to implement the PEs in the cores as they show success for most of the application domain [5]. These processing cores can be implemented in a chip to develop an MPSoC. The multi-core architecture has a number of advantages:

- Most suitable to the future process technologies as more cores will be available with advancement in technology and the complexity of the cores can be kept the same.
- Small cores can be optimized extensively.
- Computational performance scales almost linearly with the number of cores.
- Some cores can be switched off/on depending on the requirements.

- Faulty cores can be discarded to make the multi-core concept as fault tolerant.
- Multiple cores can be configured in parallel to improve the performance.
- Individual clock domain per core is possible and it is possible to do partial dynamic reconfiguration on a per core basis for the reconfigurable cores.

Several multi-core chips have been developed, suitable to different application domain [17]. These chip platforms are viable alternatives for high performance computing platforms.

2.2.1 Homogeneous Architectures

All cores present in homogeneous MPSoCs are identical. Thus, the cores can be easily replicated, making it very suitable for VLSI implementation. Further, they are easy to program as compared to their heterogeneous counterpart.

Academias often propose homogeneous MPSoCs and some of the notable ones are introduced here. Massachusetts Institute of Technology proposed 16-core Raw Architecture Workstation (**RAW**) processor architecture [18]. A 167-core Asynchronous Array of Simple Processors (**AsAP**) has been proposed by University of California at Davis [19]. The University of Texas at Austin proposes The Tera-op, Reliable, Intelligently adaptive Processing System (**TRIPS**) which uses 32 chips each containing 2-core [20]. A **WaveScalar** processor has been proposed by University of Washington which contains approximately 2K simple processing elements (PEs) arranged into 16 clusters [21]. In [22], a scalable MPSoC for next generation architectures has been proposed. This architecture is based on RISC processors and distributed memories. These chips have been developed to provide high performance.

Recently, Intel and Tilera Corporation have proposed homogeneous MPSoCs [23] [24]. In [23], Intel proposes a homogeneous MPSoC consisting of 80 cores connected by an interconnection network where each core contains two floating-point units. The interconnection network is arranged as a 10×8 2D array in 275mm^2 area that contains 80 cores and packet-switched routers, operating at 4GHz. The MPSoC is designed in 65nm technology. Intel has also announced Core i3, Core i5 and Core i7, a family of multi-core processors for desktop and mobile processors [17]. In [24], Tilera Corporation announced TILE-Gx100, the world's first 100-core general purpose processor that offers the highest performance amongst any microprocessor yet announced by a factor of four. They have also simplified many-core programming with their breakthrough Multicore Development Environment (MDE) that features rapid product deployment. Hewlett-Packard announced an MPSoC consisting of multiple MIPS cores [25].

There are lot of other chip manufactures who have announced their multi-core chips targeting for different computing domains such as scientific, embedded and general purpose computing. Some multi-core chips are listed in [17].

2.2.2 Heterogeneous Architectures

The heterogeneous MPSoCs consist different types of PEs implemented in the cores. The PEs can be GPPs, specialized PEs like DSPs, FPGA fabrics, dedicated intellectual property cores (IPs), specialized memories etc. By exploiting distinct features of different type of PEs, compute performance can be increased while at the same time power consumption can be reduced and remain flexible.

Recently, academic researchers are targeting heterogeneous MPSoCs as well. Some heterogeneous architectures proposed by academia are presented in [26], [27], [28], [29] and [30].

Nollet et al. [26] present MPSoCs containing four different type of PEs: GPPs, DSPs, accelerators and reconfigurable hardware blocks. The different PEs are interconnected by a 3×3 mesh network. Smit et al. [27] propose reconfigurable architectures where the

PEs are connected by an on-chip network and one particular architecture is named as **Annabelle**. In the Smart chipS for Smart Surroundings (4S) project [28] at University of Twente, a dynamically reconfigurable MPSoC architecture has been proposed, where a core can be either a bit-level reconfigurable unit (e.g. FPGA) or a word-level reconfigurable unit (e.g. Montium [31]), or a general-purpose programmable unit (DSP or GPP). The programmability of the cores in the reconfigurable architectures facilitate the system to be targeted for multiple application domains.

Arpinen et al. [29] present an MPSoC consisting of many Altera Nios II soft-core processors and custom hardware accelerators, where they are connected by a communication network called Heterogeneous IP Block Interconnection (HIBI). The MPSoC is created on an Altera FPGA [32]. Hristo et al. [30] propose MPSoCs containing fixed Instruction Set Architecture (ISA) processors and dedicated IP cores. These MPSoCs provide high performance for fixed targeted applications.

Two heterogeneous MPSoCs: *CHAMELEON* and *Pleiades Chips* are proposed by University of Twente and UC Berkeley, respectively. These MPSoCs target DSP algorithms. The **CHAMELEON** SoC [31] contains a general purpose processor (an ARM core), a fine-grained reconfigurable part (FPGA fabric cores) and a coarse-grained reconfigurable part (MONTIUM cores [31]). Highly regular computation patters of an algorithm are executed in reconfigurable parts and irregular parts on the general purpose processor. The **Pleiades Chips** [33] contain low power domain specific processors.

Industry is also targeting heterogeneous MPSoCs. Some industrial proposals are presented in [34], [35], [36] and [17].

In [34], STMicroelectronic presents a flexible MPSoC called **StepNP**. The MPSoC contains multiple configurable multi-threaded processors, configurable PEs and networking-oriented I/O's, where all of them are connected by an on-chip network. The MPSoC fulfills the requirement for flexibility, rapid development and end-user productivity. Leijten et al. [35] propose an MPSoC that is designed by a method known as

PROPHID. The **PROPHID architecture** contains a general purpose processor for control and low to medium-performance signal processing, and domain specific processors for high-performance signal processing. Rutten et al. [36] propose an MPSoC called **Eclipse** that contains function-specific coprocessors to execute the tasks of one or more applications at the same time in order to provide high performance.

In [17], MPSoCs proposed by several companies are listed. These MPSoCs contain varying number of PEs implemented in the cores depending upon the need of the targeted application domain. Some contemporary MPSoCs show that most of them contain application-specific PEs next to more general purpose PEs and are introduced subsequently.

CELL

The Cell Processor [37] is jointly developed by IBM, Sony and Toshiba. It is composed of one manager processor and 8 floating-point units. The manager processor is PowerPC [38] and is called *Power Processor Element* (PPE). The PPE runs an operating system for controlling all the PEs. The floating-point units are called *Synergistic Processor Elements* (SPEs) and provide the required compute performance. The Cell processor uses a high speed communication network. This MPSoC is used as the heart of *PlayStation 3* game console.

Nexperia

The Nexperia [39] is developed by Philips. It contains three processors: one MIPS PR4450 for control processing and balancing off-chip functions, and two TriMedia TM3260 DSP processors for processing the multimedia content. This MPSoC mainly target the digital television and set-top box systems.

OMAP

The OMAP [40] is developed by Texas Instruments (TI). The OMAP2 MPSoC contains an ARM11 as manager processor, a TI C55x digital signal processor for accelerating DSP processing, a 2D/3D graphics accelerator and a video accelerator. This MPSoC can be found in high end Nokia cell phones. The dual-core OMAP4 chip can be found in modern tablets such as in Kindle Fire and BlackBerry PlayBook.

Nomadik

The Nomadik [41] is developed by STMicroelectronics. This MPSoC contains an ARM926 processor and several Very Long Instruction Word (VLIW) DSP cores. The DSP cores provide acceleration for different kinds of signal processing tasks. This MPSoC is used in high end cell phones.

XPP

The Xtreme Processing Platform (XPP) [42] is developed by PACT Technologies. The platform is composed of a coarse-grain reconfigurable array, adaptive computing elements and a packet-oriented communication network. The platform is suited for large domain of applications and provides high computation performance.

AVISPA CH

The AVISPA CH [43] is developed by Silicon Hive (an incubator of Philips Research) [44]. This MPSoC contains a Base Processor (BP), two Arithmetic Complex Processors (ACPs) and a pseudo-multi-port memory. The AVISPA CH provides high processing power, flexibility and hardware acceleration, and is very suitable for base band processing of existing and emerging audio and video broadcasting standards.

Configurable Cores

Recently, many companies have begun providing configurable cores targeting different application domains. These are known as *Application Specific Instruction-set Processors (ASIPs)*. The ASIPs provided by Tensilica [45] and Silicon Hive [44] are listed in [46] and [47]. These companies provide the complete toolset to generate both the MPSoC where each PE can be customized and the corresponding software programming toolset. The motivation behind ASIPs is that tuned PEs are more efficient than general-purpose PEs, all the while providing more flexibility than ASICs [48].

Fine-grain Reconfigurable Hardware

The *Field Programmable Gate Array (FPGA)*, also denoted as fine-grain reconfigurable hardware, is becoming a viable alternative for the ASIC in embedded systems market. The FPGA provides high flexibility and low upfront cost with performance between the general purpose processor (GPP) and ASIC. There is no NRE cost as FPGAs are an off-the-shelf product. FPGA fabrics are being integrated into SoCs in order to create high performance custom logic even after manufacturing. In conventional systems, FPGA fabric is fully allocated to a single processor, which is used to accelerate compute intensive tasks [49] [50]. On the other hand, in modern systems, FPGA fabrics are integrated as some of the PEs of the MPSoC. Some research works show the potential benefits of integrating the FPGA fabrics in MPSoCs [26] [51] [52]. However, there is still a lack of general agreement on how the integrated FPGA fabrics should be used as regular PEs within the MPSoCs [53]. Additionally, FPGA vendors such as Xilinx [54] and Altera [32] also provide soft and hard IP-core blocks. These provide GPP, DSP and accelerators functionality and can be integrated to develop custom MPSoCs providing flexibility and high performance.

Intuitively, it is clear that application development and run-time management of different type PEs in heterogeneous MPSoCs is difficult as compared to their homogeneous counterpart.

2.2.3 On-chip Interconnects for the Architectures

An on-chip interconnect is required to connect the PEs of MPSoCs in order to fulfill their communication needs. Out of available interconnect options such as buses, point-to-point connections and Networks-on-Chip (NoC), it has been observed that NoC is the most efficient and highly scalable [12] [13] [14].

The term NoC is used in several contexts varying from multi-layer segmented buses to on-chip networks [55]. Several researchers focus on designing efficient NoCs, which involve several challenges. Marculescu et al. [56] describe the outstanding research problems in designing NoCs and categorize them into five categories: *application modeling and optimization for NoC communication*, *communication paradigm selection*, *communication infrastructure synthesis*, *evaluation* and *validation.* Some solutions against each problem have also been suggested. Efficient solutions to these problems need to be explored from the perspective of future NoC research.

Examples of well known research NoCs are **SPIN** [57], **AEthereal** [58], **QNoC** [59], **Xpipes** [14], **PNoC** [60], **ProtoNoC** [61], **Nostrum** [62], **MANGO** [63] and **HERMES** [64]. The design approaches for the mentioned NoCs are described in their respective references. Some other design approaches have been mentioned in [65], [66] and [67]. Bjerregaard et al. [63] provide an excellent survey of existing NoC research and their practices. Lately, several start-up companies like Sonics [68] and Arteris [69] have started to commercialize the NoC concepts with their NoC products.

The NoCs require smart routing mechanisms for efficiently transferring data from one PE to another PE of the MPSoC. Some mechanisms are presented in [70] and [71]. In [70], a novel routing scheme called *DyAD* is presented, which takes the advantages of both deterministic and adaptive routing schemes by switching between them based on the network's congestion. In [71], a multi-path routing strategy is presented that guarantees in-order data delivery by optimally spreading the traffic in the NoCs to minimize the bandwidth requirements of the network.

The NoC architecture also imposes new run-time management challenges. For example, re-routing communication, i.e. changing the communication path between source and destination PEs at run-time. Additionally, resource management algorithms need to consider the properties of the interconnect.

2.2.4 Designing Multi-Processor System-on-Chip Platforms

As mentioned earlier, MPSoCs can provide the most efficient architectural solutions for supporting different domain of applications. As a result, tools to design and simulate these systems are needed. Designing MPSoCs involves several challenges [72]. For example, the number and type of processors, size of memories, communication infrastructure and different accelerators to be considered in designing a promising MPSoC for a given set of applications. Academia and industry have proposed several design methods integrated into different tools. The design methods are either software-based or hardware-based. The software-based and hardware-based techniques design simulation and hardware platforms, respectively.

2.2.4.1 Software-based Design Techniques

The software-based design approaches provide simulation platforms for MPSoCs, which are relatively easy to design than that of the hardware platforms. However, simulation platforms provide near accurate results and sometimes take long time in simulation. Some existing works to design simulation platforms are presented in [73], [74], [75] and [76].

Benini et al. [73] propose a method to design MPSoC simulation platform called **MP-ARM**. The MP-ARM platform contains ARM processors modeled based on SystemC [77] and communication architecture compliant with AMBA bus. Paulin et al. [78] present another technique to design SystemC based MPSoC simulation platform called **StepNP**. Monchiero et al. [74] present a design framework called **GRAPES**. The framework is system-level and cycle-based, which provides flexibility and modularity maintaining high simulation speed. SystemC or C++ entities are used to model intellectual property (IP) modules. The modules are captivated into C++ objects called plug-ins and are managed by the GRAPES kernel, which is the heart of the simulation framework.

Cong et al. [75] present a methodology to automatically generate fast, cycle-true, C-based simulators for coprocessors using a high-level synthesis tool and integrate them with their simulation framework **MC-Sim**. The framework is capable of accurately simulating a variety of processor, memory, NoC configurations and application specific coprocessors. Atat et al. [76] propose a system level design approach for rapid prototyping of MPSoCs starting from Matlab/Simulink specifications.

Beltrame et al. [79] present a method that starts from the description of an application in standard sequential code. Firstly, the application is profiled in order to parallelize it, which gives minimum number of processors required for a given constraint. Then, a **StepNP** based simulation platform is designed based on the parallelized components and the number of processors. The flow is applied to an MPEG4 VGA real-time encoder for industrial case study.

Interuniversity Microelectronics Centre (IMEC) [2] presents a design flow that uses three tools. The flow is depicted in Figure 2.3. The tools used are *i) CleanC* for source code cleaning, *ii) Multi-Processor Assist (MPA)* for parallelization of sequential code and *iii) Memory Hierarchy (MH)* for performing scratchpad memory management. The *CleanC* permits designers to write a high-level sequential code that is optimized for parallelization. The *MPA* tool empowers designers to extract the potential parallelization present in an application using the functional and data parallelism. The *MH* tool automatically schedules data transfers between main memory and local memory by analyzing the input source code.

The designed MPSoCs based on simulation framework are being targeted by many researchers as they are easy to design and need less design-time.

2.2.4.2 Hardware-based Design Techniques

Hardware platforms are difficult to design as compared to the simulation platforms. However, hardware platforms provide faster execution than the simulators. Some methods to

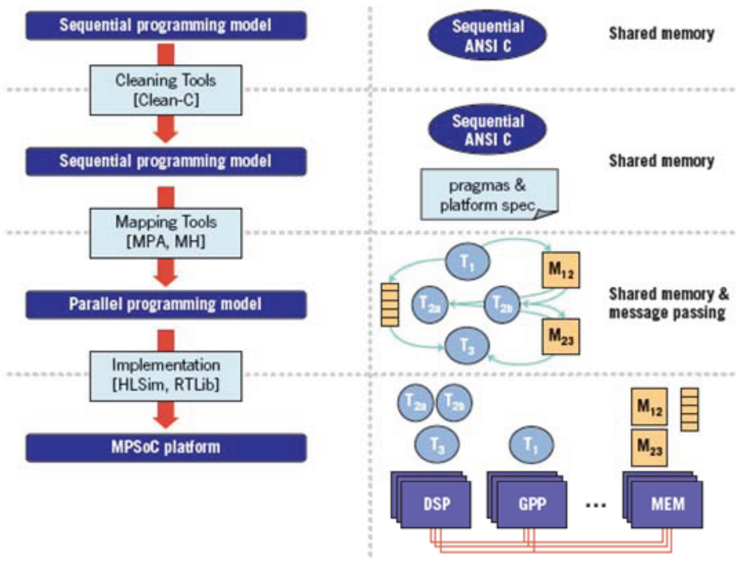


Figure 2.3: IMEC MPSoC design flow [2]

design hardware platforms are presented in [80], [81], [82], [83] and [84].

Nikolov et al. [80] present a methodology implemented in a tool called *Embedded System-level Platform synthesis and Application Mapping (ESPAM)* for automated design, programming and implementation of MPSoCs. The methodology considers an application, system-level platform, and mapping specifications as input for performing the automation. The proposed methodology is evaluated by automatic generation and programming of several MPSoCs for executing real-time applications. STMicroelectronics [81] propose a low-cost modular approach for generating hardware platforms that offer design and verification of complex MPSoCs.

Atienza et al. [82] present a framework to design MPSoCs on FPGA. The designed MPSoC provides speed-ups of three orders of magnitude as compared to cycle-accurate MPSoC simulators. Sun et al. [83] propose a methodology for MPSoC synthesis. The methodology is implemented using the Xtensa platform from Tensilica Inc [45]. The methodology is evaluated by automatically generating custom MPSoCs for several complex embedded software benchmarks and the results show that synthesized MPSoCs provide faster results. Kumar et al. [84] present a methodology implemented in a tool called *Multi-Application Multi-Processor Synthesis (MAMPS)* for generating MPSoCs in a systematic and fully automated way for multimedia applications. The tool generates MPSoCs for Xilinx FPGAs. For scalable architectures, techniques to generate NoC-based MPSoC on FPGA are presented in [85] [86] and [87].

So far, techniques to design MPSoCs have been described, which design either simulators or hardware platforms. Some of the techniques design general MPSoCs to be targeted to map applications at later stages [73] [76] [81], while others application specific MPSoCs [79] [2] [80] [83] [84]. An application specific MPSoC is customized to serve a fixed application. To handle dynamism such as adding a new application into the system at run-time, general MPSoCs are required.

In order to handle modern complex applications requiring large number of processors, the MPSoCs need to be NoC-based for efficiency and scalability. However, it has been observed that most of the design methods produce MPSoCs that are not NoC-based. Further, it has been observed that MPSoCs on Field Programmable Gate Arrays (FPGAs) for hardware platforms are a new and increasingly important trend [80] [82] [84] [85] [86] [87]. Dorta et al. [88] provide a nice overview of the MPSoCs on FPGAs. These facilitate rapid prototyping and allow for research in new architectures without the worries of their ASIC production. However, these have reduced performance compared to their ASIC counterpart but offer several advantages like flexibility, reconfiguration, less time-to-market and less cost, to compensate for the same. Modern FPGAs can accommodate 80-100 soft-core processors in a single chip and NoC is the best solution to manage such large number of cores [89]. Techniques are also available to design thousand core systems by using multiple FPGAs and such a system has been referred to as *Research Accelerator for Multiple Processors (RAMP)* [90].

2.3 Mapping Applications on Multi-Processor System-on-Chip Platforms

Before starting the mapping of applications on MPSoC platforms, the applications need the following processing:

- Parallelization of the application, adding synchronization and inter-task communication in the parallelized tasks, and management of the memory hierarchy communication. This job can be furnished by state-of-the-art application parallelization tools [91] [2]. The parallel tasks can be executed on different platform resources concurrently in order to accelerate the application execution.
- Checking the parallelized code and making sure that it is functionally correct and optimized for a given set of platform parameters.
- In the case of heterogeneous platforms, a task binding process is required. For each task, the binding process defines the processor types on them the task can be mapped. It also specifies the cost of mapping on the different processor types.

Mapping tasks of applications on an MPSoC platform involves assignment and ordering of the tasks and their communications onto the platform resources in view of some optimization criteria like reducing energy consumption, improving compute performance etc. The optimization is necessary to satisfy performance constraints of the applications. Therefore, efficient mapping techniques are required in order to optimize the performance. The mapping techniques need following number of parameters:

- An application model (e.g., Task Graph [92], Data Flow Graph [93] etc.).
- An architecture model of the MPSoC platform (e.g., topology, number of PEs and their type, interconnection scheme etc.).

- The constraints of the application (e.g., compute performance and/or power requirements etc.).
- The performance model of inter-process communication (execution time, energy consumption etc.).
- An estimate of the worst case execution time of the process implementations on different PEs.

The mapping problem is being addressed by several researchers who communicate their views through various forums such as ArtistDesign Network of Excellence [8], International Forum on Embedded MPSoC and Multicore [7], various International conferences and journals. The mapping techniques have been developed by targeting specific application domain for the most promising MPSoC architecture.

Mapping applications' tasks on MPSoC platform resources can be accomplished at either design-time (static) or run-time (dynamic). Design-time mapping techniques consider predefined set of applications with known computation and communication behavior and a static platform. Therefore, they are not suitable for dynamic workloads such as adding a new application into the platform at run-time. Dynamic (run-time) mapping techniques are required for scenarios where application tasks need to be loaded into the platform at run-time. After mapping tasks, task migration can be used to revise placement of some of the already executing tasks, if the user requirement is changed or a new application has entered into the system.

In next subsections, we discuss the design-time and run-time mapping techniques reported in the literature. Our work focus mainly on run-time mapping, so we explore run-time mapping techniques extensively and provide some introduction of design-time mapping techniques.

2.3.1 Design-time Mapping

Design-time mapping techniques have a global view of the system which helps in making better decision for using the system resources. Thus, a better quality of mapping may be achieved as compared to the run-time mapping techniques that are restricted normally to a local view. Most of the mapping related works in literature cover design-time mapping techniques.

Design-time mapping techniques for bus-based and NoC-based MPSoCs are presented in [94] [95] and [96] [97] [98] [99] respectively. The bus-based architectures are not scalable and thus they enforce scalability issues to their related mapping techniques. Hu et al. [96] propose a mapping technique called *Communication Weighted Model (CWM)* and show that the overall power consumption is reduced by decreasing the energy consumption in communication. Marcon et al. [97] extend the work in [96] and propose a technique called *Communication Dependence and Computation Model (CDCM).* The CWM considers only the communication volume, whereas CDCM considers the volume and timing of the communication. Murali et al. [98] present a methodology that maps multiple use-cases onto NoC architecture and performance constraints for each use-case are satisfied. Rhee et al. [99] investigate *core-switch mapping (CSM)* problem that optimally maps cores onto NoC architecture in order to minimize energy consumption or NoC congestion.

Different well established search approaches are also used to develop design-time mapping techniques in order to find optimal placement of tasks on platform PEs. Genetic approach is used in [100] [101], Tabu Search in [102] and Simulated Annealing in [103] [104]. Lei et al. [100] present a two-step genetic mapping algorithm aimed at optimizing the application execution time. Wu et al. [101] present a genetic mapping algorithm that uses *Dynamic Voltage Scaling (DVS)* to reduce the energy consumption. Manolache et al. [102] investigate the task mapping aimed at guaranteed network packet latency in order to guarantee for worst case application response time. Orsila et al. [104] propose a Simulated Annealing algorithm in order to optimize execution time and memory consumption, whereas traditional approaches only focus on the execution time.

Some other recent design-time mapping techniques are summarized in [105], [106], [107] and [108].

All the design-time techniques find placement of tasks at design-time. Therefore, these techniques are not suitable for run-time varying workloads in the systems, which require re-mapping/run-time mapping of applications (e.g. networking and multimedia applications). Even if these mapping techniques are inadequate for the dynamic workload scenarios, such techniques might be useful to find the initial task placement, or be optimized to be working at run-time.

2.3.2 Run-time Mapping

In contrast to the design-time mapping, run-time mapping needs to consider the time taken to map each task as it contributes to overall application execution time. Furthermore, the tasks are mapped one by one, unlike the static case where all the tasks are mapped at once by looking globally at the system. Therefore, greedy algorithms are used for efficient mapping in order to optimize performance metrics such as energy consumption, communication latency, execution time etc.

In addition to the suitability of the run-time (dynamic) mapping techniques over design-time (static) techniques in the case of dynamic workload scenarios, they also offer a number of other advantages:

- *Adaptability to the available resources*: The available resources vary over time as the applications of the dynamic workload scenario enter at run-time.

- *Ability to enable unforeseeable upgrades*: It is possible to upgrade the system for new applications or changing standards that are not known at design-time, even after the delivery of the system to the end-user.

- *Ability to avoid defective parts of a SoC*: If one or more processing cores are not operating properly after production of a SoC, then the defective cores can be disabled before the mapping process [109]. Aging can lead to defective cores that are unforeseeable at design-time.

The run-time mapping techniques allocate tasks and their communications to platform PEs and interconnect links respectively for all the applications to be mapped. When the mapped applications start execution, the mapping of one or more running applications needs to be reconsidered in case of following events:

- When a new application is entered into the system and it needs resources from the already executing applications.

- When parameters of a running application is modified.

- When a running application needs to be killed in order to free it's occupied resources.

- When the performance requirements of a running application are changed. This might need extra resources for performing extra functionality.

- When current mapping is not sufficiently optimal, it requires (re-)mapping.

The aforementioned issues can be handled only by run-time mapping techniques as the issues are dynamic and need to be handled at run-time.

At run-time, mapping of new applications to be supported onto a platform can be handled either by performing all the processing at the same time, i.e. on-the-fly processing or by using previously analyzed results. For on-the-fly processing, efficient heuristics are required to assign new arriving tasks on the platform resources. These heuristics cannot guarantee that strict timing deadlines are met due to limited processing resources at run-time. However, such heuristics are applicable to any platform as they do not use any platform specific analysis results computed in advance. For mapping using previously design-time analyzed results, light weight run-time mapping heuristics are required as the compute intensive analysis is performed at design-time. Such heuristics map applications more efficiently than on-the-fly heuristics, but the analysis results will not be applicable to all the platforms. Next, we discuss on-the-fly mapping heuristics and mapping strategies that uses design-time analysis results, reported in the literature.

2.3.2.1 On-the-fly Mapping

The mapping techniques target homogeneous or heterogeneous MPSoCs depending upon the requirement of applications.

Techniques Targeting Homogeneous MPSoCs

Some run-time mapping techniques targeting homogeneous MPSoCs are presented in [110], [111], [112], [113], [114], [115], [116], [117], [118] and [119].

Chou et al. [110] propose a technique that incorporates the user behavior information in the resource allocation process; that allows system to better respond to real-time changes and adapt dynamically to user needs. This consideration saves 60% communication energy when compared to an arbitrary task allocation technique. Peter et al. [111] present a heuristic algorithm that is distributed over the processors and thus can be applied to systems of random size. Also, tasks added at run-time can be handled without any difficulty, allowing for inline optimization. Task migration takes place based on local information on processor workload, task size, communication requirements, and link contention. The mapping results for several example task sets show that quality achieved by the presented algorithm is within 25% of that of the exact algorithm, for a 3×3 processor array. Briao et al. [112] present strategies based on bin-packing algorithms for running soft real-time applications. They combine different types of algorithms to get better allocation results. In order to save energy, the system turns off idle processors and applies *Dynamic Voltage Scaling (DVS)* to processors with slack. Chou et al. [113] propose a technique that considers multiple PEs operating at multiple voltage levels for energy aware mapping.

Ngouanga et al. [114] describe a technique based on attraction forces between the communicating tasks. The technique tries to place such tasks close to each other on the MPSoC PEs in order to reduce communication overhead. Mehran et al. [115] present a *Dynamic Spiral Mapping (DSM)* heuristic algorithm for 2-D mesh topologies where placement for a task is searched in a Spiral path, trying to place the communicating tasks close to each other. Before starting the spiral search, degree for each task in an application is found and the task having maximum degree is placed at the center of the mesh to facilitate the closer mapping of communicating tasks. Sassatelli et al. [116] propose two different techniques called *proactive* and *reactive* communications and conduct cycle-accurate evaluations of these techniques. Authors state that cycle-accurate homogeneous systems may become a viable alternative in near future bringing the benefits such as high performance, low power consumption and run-time load balancing. In [118], run-time management problem is phrased as a multi-dimensional multiple-choice knapsack (MMKP) problem. Moreira et al. [119] present a technique that first assigns tasks to virtual cores (VCs) while trying to minimize total number of VCs and total bandwidth used. Thereafter, the VCs are mapped to real cores.

Techniques Targeting Heterogeneous MPSoCs

These days MPSoCs are highly heterogeneous for better fulfilling the application's requirements. In case of heterogeneous MPSoCs, the task binding process is realized before

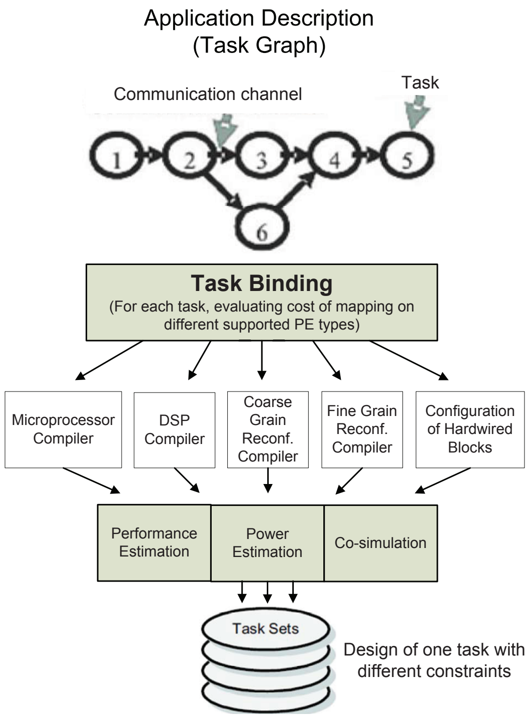

Application Description
(Task Graph)
Communication channel
Task
1
2
3
4
5
6
Task Binding
(For each task, evaluating cost of mapping on different supported PE types)
Microprocessor Compiler
DSP Compiler
Coarse Grain Reconf. Compiler
Fine Grain Reconf. Compiler
Configuration of Hardwired Blocks
Performance Estimation
Power Estimation
Co-simulation
Task Sets
Design of one task with different constraints

Figure 2.4: Task binding process

starting the mapping. For each task, the binding process defines the PE types on them the task can be mapped along with the cost of mapping. Figure 2.4 shows the binding process (design-time profiling) for an example application, where the application tasks are analyzed on different type of PEs such as GPP, DSP, coarse grain reconfigurable hardware etc. The profiling provides performance, power and resource utilization for each task on different type of PEs.

Smit et al. [51] present an algorithm that first maps tasks needing scarce resources and then all other tasks by taking availability of the platform resources into account. More techniques to map streaming applications onto multi-core architectures are presented in [27]. In the Smart chipS for Smart Surroundings (4S) [28] project, a spatial mapping tool named *SMIT* is developed. The *SMIT* is excited by a RTOS when an application needs to be mapped on the MPSoC. The tool (spatial mapper) takes *description of system architecture*, *functional description of application*, *process realizations*, *current*

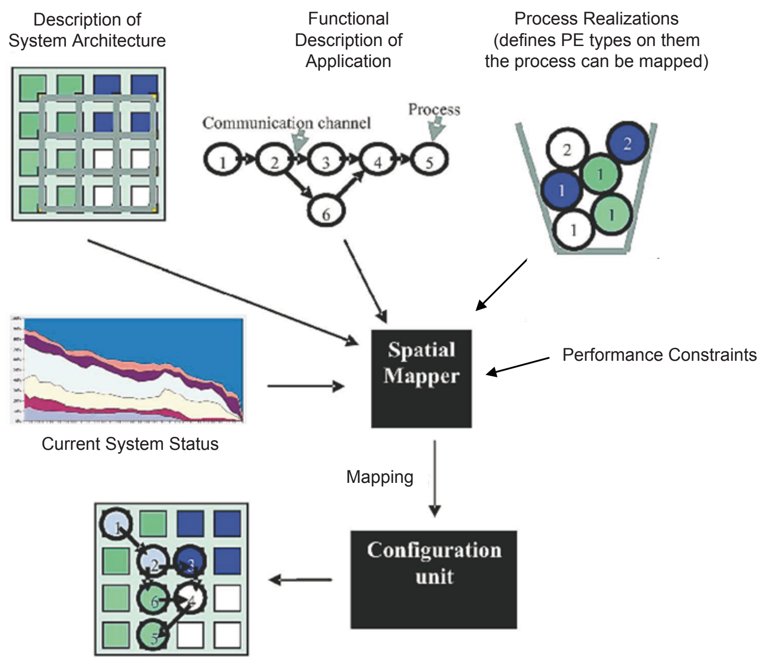

Description of System Architecture
Functional Description of Application
Process Realizations (defines PE types on them the process can be mapped)
Communication channel
Process
Spatial Mapper
Performance Constraints
Current System Status
Mapping
Configuration unit

Figure 2.5: Run-time mapping followed by SMIT Mapper tool

system status and *performance constraints* as input and outputs a *mapping* providing placement of processes and communication channels in the system architecture, which is used to configure the system, as shown in Figure 2.5. The *process realizations* are the obtained by applying the task binding process described earlier. The *SMIT* tool performs optimization over all the system PEs and communication network.

Holzenspies et al. [120] present a run-time spatial mapping technique consisting of four steps to map streaming applications onto a heterogeneous MPSoC. The algorithm is implemented on an ARM926 running at 100 MHz and it takes less than 4 ms to run the HIPERLAN/2 example. Braak et al. [121] propose another run-time spatial mapping technique that spans both the task graph and the MPSoC platform to find optimal mapping of tasks. Nollet et al. [26] describe a run-time task assignment heuristic for efficiently mapping the tasks in an MPSoC containing FPGA fabric tiles. With the presence of FPGA fabric tiles, the heuristic is capable of managing a configuration hierarchy and improves the task assignment success rate and quality.

Faruque et al. [122] present a run-time agent based distributed application mapping technique targeting large MPSoCs such as 32×32 and 32×64 systems. For large MPSoCs, the distributed mapping technique is better than the state-of-the-art run-time mapping techniques that use *Centralized Manager (CM)* approach. The CM approach for large MPSoCs may face the following problems:

- Single point of failure.
- Large volume of monitoring-traffic by the CM.
- High computational cost to calculate mapping inside CM.
- Bottleneck around the CM as every core sends its status to the CM after every instance of mapping. Thus, the CM becomes a hot spot.

The distributed technique reduces the monitoring traffic and computational effort.

Schranzhofer et al. [123] propose a polynomial-time multiple-step heuristic consisting of initial solutions followed by task re-mapping algorithms considering power constraints. First, initial solutions for power-aware scenario are derived, and then task re-mapping is performed to improve the solutions. Lei et al. [124] present a two-step Genetic Algorithm that finds placement of tasks onto the available cores aiming at minimization of the overall execution time. Becchi et al. [125] propose a mechanism where benefits of heterogeneous cores are bolstered by exploiting thread migration between the cores. The cores implement combination of Alpha EV5 and Alpha EV6 processors. By appropriate mapping and then migration of different threads to heterogeneous processor cores, the resource utilization is maximized.

Theocharides et al. [126] demonstrate a system-level bidding-based task allocation strategy that provides significant performance improvements when compared to a round robin allocation. The obtained results motivate for further investigation of system level optimization. Schneider et al. [127] propose a placement methodology based on a hierarchical application and infrastructure description. The methodology places the application components with higher communication demands close to each other in order to minimize the overall communication costs. This also prevents blocking of long low-performance communication links. Huang et al. [128] introduce self adaptability to a run-time task allocation technique, which is achieved by dynamically adjusting a set of key parameters based on current resource utilization. Carvalho et al. [129] present heuristics where tasks are mapped on-the-fly according to the communication requests and the load in the NoC links.

At run-time, some mapping techniques use task migration to migrate tasks from one PE to another when performance bottleneck is detected or when the workload needs to be distributed more homogenously in the whole system [112] [111] [125]. Task migration may also be used in case user requirement is changed or a new application has entered into the system in order to revise the placement of some of the presently executing tasks. In migration, the tasks should be migrated without completely stopping and restarting the already executing applications.

Some task migration mechanisms are presented in [130], [131] and [112]. The mechanism in [130] uses *task migration points* as a point of reference for migrating a task from one PE to another. Authors in [131] use *checkpoints*, to define the point of reference. In [112], migration is based on a *copy* model. Issues related to the task migration such as the cost to interrupt a given task, saving its context, transmitting all of the data to a new PE and restarting the task in the new PE are discussed in [130], [131] and [132].

2.3.2.2 Based on Design-time Analysis Results

Mapping strategies based on design-time analysis results perform compute intensive analysis at design-time and use the analyzed results at run-time. This facilitates for a light-weight run-time platform manager that dynamically and efficiently maps the applications based on status of the platform resources.

Design-time analysis is performed by taking application description, platform specifications and design objectives into account in order to explore design points to be used at run-time. The design points contain tasks to PEs combinations, i.e. mappings, representing trade-offs between different performance metrics. Exploring all the possible tasks to PEs combinations exhaustively is not feasible within a limited time. Therefore, faster analysis strategies having some design objectives are required to explore efficient mappings.

Most of the design-time analysis techniques reported in literature provide a single mapping for the application. Design-time mapping techniques to find a mapping for an application reported in Section 2.3.1 can be used to accomplish such analysis. Some other such analysis techniques are presented in [133], [134], [135], [136], [137] and [138]. They perform exploration in view of some optimization parameters such as computational performance and energy. The explored single mapping cannot handle dynamism in resource availability and performance requirement at run-time.

Design-time analysis strategies that generate multiple mappings for the application have recently been reported in [139], [140], [141], [142], [143], [144], [145] and [146]. The generated mappings can be used to handle dynamism in resource availability and performance requirement at run-time. In [139] and [140], exploration is performed in view of optimizing for power consumption and performance in order to identify the best performance/power trade-offs. Stuijk et al. [141] optimize for resource usage. Beltrame et al. [142] optimize for energy and delay. They try to minimize number of simulations required to identify the mappings providing energy/delay trade-offs. Palermo [143] present a multi-objective exploration framework that also provides energy/delay trade-off points. In [144] and [145] too, authors present multi-objective exploration approach. Jia et al.

[146] present an infrastructure called *NASA (Non Ad-hoc Search Algorithm)*, which uses different combination of search strategies to explore the mapping.

There has been quite some research in multiple applications DSE. Some researchers focus on scenario based approach where multiple application mapping scenarios are explored at design-time in order to handle dynamism in number of active applications at run-time [141], [147], [148]. A scenario contains a set of simultaneously active applications. The scenarios have also been referred to as use-cases [84] [98] [149]. The scenario based approaches are not scalable as the number of scenarios increases exponentially with the number of applications, which might become intractable.

A few strategies that perform mapping using design-time analysis results are presented in [150], [151], [152] and [153]. In [150], analysis result includes only a single mapping having minimum average power consumption. In [151] and [152], analysis results include multiple mappings having trade-off in terms of target power consumption and performance. In [153], design-time analysis gives ideal PE count and memory required for current state of the application. The design-time analysis results have been used by run-time platform manager in order to map applications on the platform. The manager invokes run-time selection strategy to select the best mapping from the design-time analyzed mappings in order to configure the applications on the platform resources.

2.3.3 Analysis of Mapping Techniques

So far, in this section, relevant existing techniques for mapping applications on MPSoC platforms have been described. The aim of this sub-section is to critically examine the existing techniques and identify avenues of research. The criteria for evaluation are suitability for dynamic workloads in the systems, gain in different performance metrics and computational complexity for scalability.

Design-time mapping techniques described in Section 2.3.1 consider a fixed set of applications and a static platform as input and thus find mapping by having a global view of the system. Therefore, they may provide better quality of mapping as compared to run-time time techniques having normally a local view. However, these techniques are not suitable for dynamic workload scenarios such as adding a new application into the system at run-time. Run-time mapping techniques are required to handle such scenarios.

Run-time mapping techniques described in Section 2.3.2 load the application tasks into the system at run-time as and when the application need to be supported. These techniques split into two directions. Some tackle the mapping problem by defining efficient heuristics described in Section 2.3.2.1, where new arriving tasks are assigned on the system resources at run-time and all the processing is done at the same time, i.e. on-the-fly. These on-the-fly heuristics cannot guarantee for schedulability, i.e. for strict timing deadlines due to limited processing power at run-time. Others analyze applications at design-time (offline) by defining efficient analysis strategies described in Section 2.3.2.2, where schedules and allocations, i.e. mappings are computed that are then stored onto the system. In order to support an application at run-time, the best mapping is selected from the stored mappings based on the required performance and available system resource, which is then, used to configure the system. This facilitates for a light weight run-time system manager and map applications more efficiently than on-the-fly heuristics. However, flexibility in these approaches is limited, since all potential applications must be known in entirety at design-time and analysis results will be applicable only to the analyzed platform. Therefore, design-time analysis needs to be repeated when the application set or platform changes. Further, storing analysis results introduces additional overhead. In contrast, on-the-fly heuristics are applicable to any application set and any platform.

It has been observed that the mapping techniques target heterogeneous MPSoCs for better fulfilling the application's requirements as compared to their homogeneous counterpart. Further, the MPSoCs are based on NoC communication infrastructure for scalable and efficient architectures.

At the start of this research we witnessed that most of the existing on-the-fly heuristics for NoC-based heterogeneous MPSoCs support only a single task on each PE, for example, heuristics in [51], [27], [120], [26], [122] and [129]. Supporting a single task on each PE is not a realistic scenario. Additionally, performance provided by the existing mapping techniques is still a concern mainly due to communication bottlenecks. Therefore, better mapping techniques should be investigated to provide better performance and the techniques should be extended to MPSoCs containing PEs supporting multiple tasks. Chapter 3 deals with our proposed run-time mapping techniques based on a packing strategy targeting MPSoC containing single task supported PEs and we show that the performance is improvement. Chapter 4 deals with the extension of the proposed techniques to MPSoCs containing multiple tasks supported PEs for realizing realistic scenarios. The extended techniques based on the packing strategy do not consider communication between tasks during mapping and therefore they need further investigation in order to achieve better performance. Chapter 5 deals with the communication-aware mapping techniques where communicating tasks are mapped on the same PE as far as possible in order to reduce communication overhead, leading to significant performance improvements.

The communication-aware mapping techniques proposed in Chapter 5 and reported in literature such as in [127], [115], [114] and [129] do not consider computation load balancing while reducing communication overhead. So, these techniques do not perform well enough for the scenarios where computation overhead dominates communication overhead or both the overheads become significant. Therefore, computation and communication aware mapping techniques should be investigated to take care of both the overheads. This will be at the cost of managing both the overheads at the same time. Chapter 6 deals with such techniques, delineating substantial performance improvements.

Design-time analysis strategies take application and platform specifications as input and explore mappings with some design objectives (exploration objectives) as shown in

Figure 2.6. The explored mappings (operating points) provide guidelines for configuring the application at run-time, which is shown as run-time guidelines. The same analysis strategies can be applied to all the applications one after another, which might need to be supported into the system at run-time, as shown in Figure 2.6. Existing analysis strategies reported in Section 2.3.2.2 have several drawbacks:

- The analysis results are applicable only to the pre-analyzed fixed platform. Therefore, analysis needs to be repeated with any changes in the platform. In [146], exploration is performed for multiple platforms, so the applicability gets extended. However, it is limited to the set of explored platforms.

- They do not provide optimal mappings from throughput point of view in some cases. These strategies perform optimization for some performance metrics like energy, resource optimization etc. and in turn map the potentially parallel executing tasks on the same PE, forcing their execution in sequence. This often reduces the available parallelism, thereby reduced throughput.

- They evaluate large number of mappings for relatively larger platforms including some duplicate mappings and thus do not scale well with the platform size. Therefore, they are not suitable for advanced available commercial platforms containing hundreds of PEs [23] [24] and for anticipated MPSoCs [9] containing thousands of PEs. The duplicate mappings just differ in placement of tasks on different PEs with the same tasks to PEs allocation and provide the same throughput.

- The analysis results might not include mappings satisfying the constraints in case of limited resources. At run-time, this case forces the application to be put into a relaxed application set and it is not mapped immediately, which may result in missing the strict timing deadline.

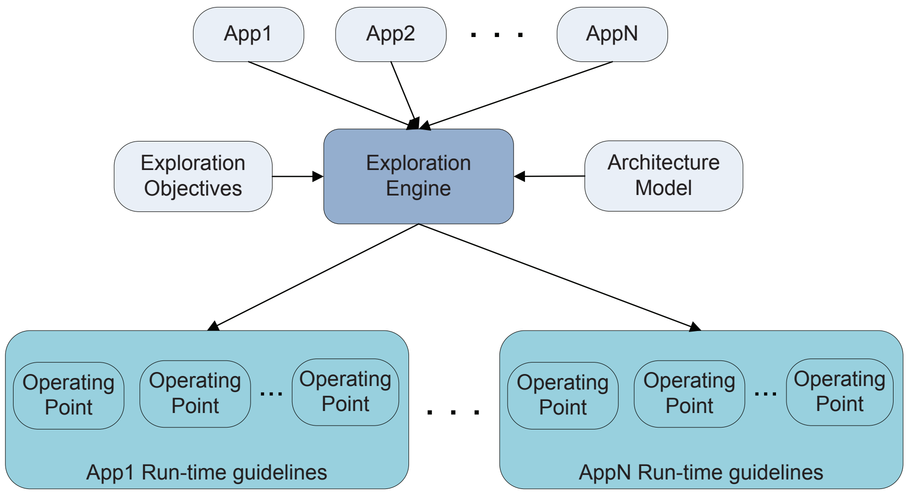


Figure 2.6: Design-time analysis of applications

To overcome the above mentioned problems, there is a need to investigate efficient analysis strategies. Chapter 7 deals with faster and efficient analysis strategies, overcoming the problems.

In existing approaches, design-time analysis of multiple applications at the same time has been handled by scenario based approaches, where a scenario represents number of active applications at run-time. These approaches are not scalable as the number of scenarios increases exponentially with the number of applications. Therefore, simultaneously active applications can be mapped one after another to overcome the scalability issues.

At run-time, the platform manager need to select the best mapping from the stored analysis results based on the desired performance and available resources. Chapter 7 deals with run-time mapping heuristic that perform efficient selection of the best mapping, which is then used to configure the platform. In case a new application need to be supported for which analysis results are not available, on-the-fly heuristics can be used.

2.4 Summary

An in-depth literature survey on various multiprocessor architectures has been presented. These architectures can be homogeneous or heterogeneous. Heterogeneous architectures provide better performance by exploiting the distinct features of the different type of PEs present in the architecture. However, they are difficult to program as compared to their homogeneous counterpart.

Different communication infrastructures to fulfill the communication needs of the PEs present in the MPSoC are described. It has been shown that NoCs are the future communication infrastructure as these have several advantages over others such as scalability and efficiency. However, the NoC architecture imposes some new challenges. For example, mapping techniques need to consider properties of the NoC as well.

A detailed review on available design-time and run-time mapping techniques are presented. Their advantages and disadvantages for different type of workload scenarios are described. For dynamic workload scenarios, run-time techniques are proven to be more prevalent and useful. Additionally, they offer several other advantages over design-time techniques such as ability to enable unforeseeable upgrades, ability to avoid defective parts of a SoC etc. The mapping techniques are extensively analyzed to identify their strengths and weaknesses. Based on the analysis, avenues of research have been identified, which will be explored later in the thesis.

Chapter 3

Mapping Single-Task-per-Processing-Element

Most of the existing run-time mapping techniques for NoC-based heterogeneous MPSoCs consider a single task on each processing element (PE), for example, the techniques in [51], [27], [120], [26], [122] and [129]. The performance provided by the existing techniques is still a concern mainly due to communication bottlenecks. The Nearest Neighbor (NN) and Best Neighbor (BN) techniques proposed in [129] try to reduce communication bottlenecks up to some extend for certain scenarios. However, they are not efficient for the scenarios where applications with varying number of tasks are considered.

In this chapter, we propose and evaluate three run-time mapping techniques for efficient mapping of applications onto NoC-based Heterogeneous MPSoCs. The techniques attempt to map the tasks of an application in close proximity in order to minimize the communication overhead. In addition, they have been shown to alleviate NoC congestion bottlenecks to maximize overall performance. Based on our evaluations to map applications with varying number of tasks onto an 8×8 NoC-based MPSoC, we demonstrate that the techniques are capable of reducing total execution time of the applications along with the average channel load and average packet latency in the NoC when compared to state-of-the-art run-time mapping techniques. A preliminary version of this work has been published in [C-1].

The rest of the chapter is organized as follows. The conceptual MPSoC architecture used in this work is described in Section 3.1. We present a strategy called packing for efficient application mapping in Section 3.2. Efficient run-time mapping heuristics based on the packing strategy are presented in Section 3.3. Experimental results and their comparisons are discussed in Section 3.4. Section 3.5 summarizes the chapter.

3.1 NoC-based MPSoC Architecture

The MPSoC architecture used in this work contains a set of processing nodes which interact via a communication network [64] composed of routers (R) as shown in Figure 3.1. Each processing node can support either a software task or a hardware task. Software tasks execute in instruction set processors (*ISPs*) and hardware tasks execute in reconfigurable logics (*reconfigurable areas-RAs*) or in dedicated IP-cores (*IPs*). Induction of RAs in the platform facilitates flexibility to hardware at a similar level to the software (*ISPs*) for its programmability. The communication network [64] has a 2-D mesh topology that uses wormhole packet switching, handshake control flow, input buffers and deterministic XY routing algorithm. In the XY routing, first the packet is transferred in X-direction and then in Y-direction for transferring packets from one processing node to another processing node. For inter-task communication, message passing protocol is used, which is similar to one described in [129].

Among the available processing nodes, one of them is used as the Manager Processor (M) that is responsible for *task mapping*, *task scheduling*, *resource control* and *configuration control*. The configuration overhead results are used to simulate the *configuration control* process [154]. In heterogeneous MPSoCs, *task binding* is required before *task mapping*. For each task, the binding process defines platform resource types on which the task can be supported. For example, defining ISPs for software tasks and RAs for hardware tasks. *Task scheduling* uses a queue strategy and there are three queues, one

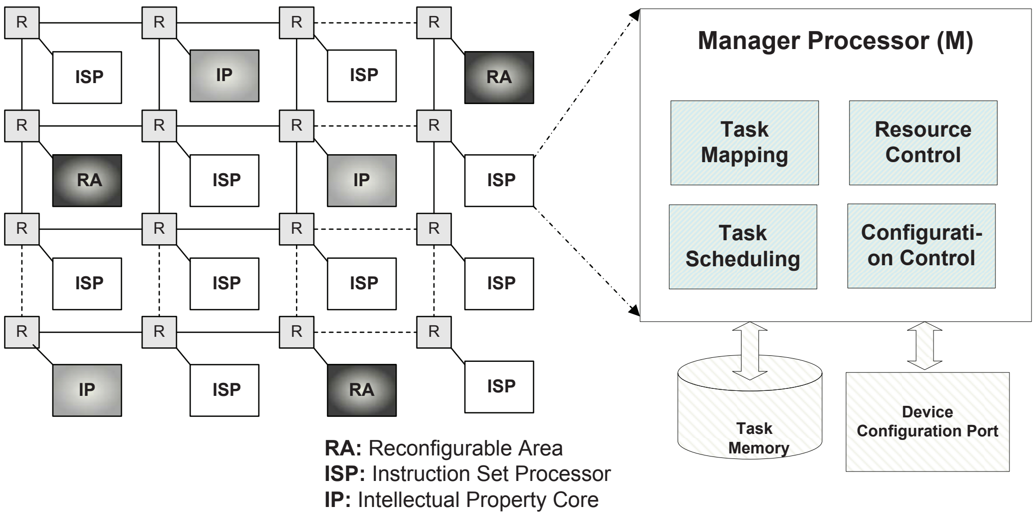


Figure 3.1: Conceptual MPSoC architecture

for each type (i.e. hardware, software and initial) of task. These task types are defined in the next section. Initial task is the starting task of an application that is mapped first. A task enters into its corresponding queue (hardware, software or initial) if there is no free supported resource in the platform. The task waits in the queue until a resource of the same type is not available in the platform.

The Manager Processor knows only the initial tasks for each application. Once, the initial tasks are mapped and their execution is started, the communication requests are sent to the communicating tasks at run-time and they are loaded into the MPSoC platform from the *task memory* if they are not already present in the platform. For *resource control*, the resources' status is updated at run-time to provide the Manager Processor with an accurate information about the resource occupancy as task mapping decision needs to be taken based on the PEs and NoC usage.

3.2 The Mapping Strategy

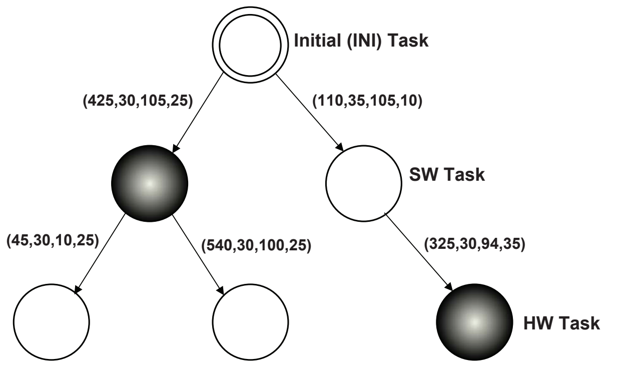
In this section, we present an improved approach for mapping tasks of applications in a systematic manner. First we introduce some definitions necessary for proper understanding of the approach. Then, we discuss the techniques to find the placement of initial tasks of applications followed by the improved approach (packing strategy) for efficiently mapping rest of the tasks.

3.2.1 Definitions

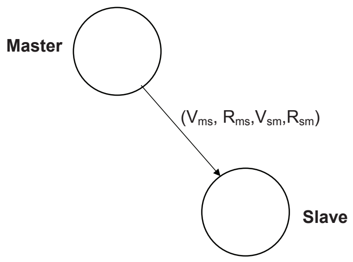
The definitions necessary to explain our mapping approach are as follows:

- `Application task graph`: It is represented as an acyclic directed graph $TG = (T, E)$, where T is set of all tasks of an application and E is the set of all edges in the application. Figure 3.2 (a) describes an application having initial (INI), software (SW) and hardware (HW) tasks along with the edges (E) connecting these tasks. A connection (edge) between two tasks defines master-slave pair (communicating task) as in Figure 3.2 (b), i.e., a connection contains master and salve tasks. Initial task has no master. A task $t_i \in T$ is represented as (t_{id}, t_{type}, t_{exec}), where t_{id} is the task identifier, t_{type} is the task type (hardware, software, initial) and t_{exec} is the task execution time. E contains all the pair of communicating tasks and is represented as $(mt_{id}, st_{id}, (V_{ms}, R_{ms}, V_{sm}, R_{sm}))$, where mt_{id} represents the master task identifier, st_{id} represents the slave task identifier; V_{ms} and R_{ms} are the data volumes and data rate respectively sent from master to slave (ms); V_{sm} and R_{sm} are the data volumes and data rate respectively sent from slave to master (sm). The message rates (R_{ms}, R_{sm}) are described as percentage of available link bandwidth. As mentioned earlier, deterministic XY routing algorithm is used to transmit and receive the messages, and both rates are relevant in the model as the path taken by messages may be different.

- `MPSoC architecture`: A NoC-based heterogeneous *MPSoC architecture* is represented as a directed graph $AG = (P, V)$, where P is the set of tiles and V is the



3.2.a: Application Modeling with INI, SW, HW tasks and edges



3.2.b: Master-Slave pair representing master and slave tasks

Figure 3.2: Application modeling and master-slave pair

set of physical channels between the tiles. A tile $p_i \in P$ contains a router (R), a network interface, a processing element (PE), local memory and a cache. A router is represented as R = (p_{add}, p_{type}), where p_{add} represents the unique PE address used to receive packets and p_{type} represents the type of PE (hardware or software) connected to the router. When tasks get mapped on a tile, the tile gets associated with following additional attributes: tasks mapped on the tile represented as p_{tasks}, the number of tasks mapped on the tile represented as $p_{tasksnum}$ and the capacity of tile showing the maximum number of tasks it can support represented as p_{cap}. If $p_{tasksnum}$ reaches to p_{cap} then no further task can be mapped on the tile. In this chapter, single task supported PEs are considered, so the task set p_{tasks} can contain maximum one task and maximum value of $p_{tasksnum}$ can be one as the value of p_{cap} is set to one. Each physical channel $v_{i,j} \in V$ keeps the *channel width* information in bits and *available bandwidth usage* (% of available bandwidth) for transmission of data.

- `Mapping`: The task mapping is represented by function *mpg*: $t_i \in T \longmapsto p_i \in P$, which maps a task of an application to a tile in the MPSoC platform. Task mapping is activated when a mapped task need to communicate with a not yet mapped task at run-time.

3.2.2 Placing Initial Tasks

Initial tasks are considered as software tasks and hence these are mapped onto software processing elements. Initial tasks placement has significant impact on the performance of run-time mapping techniques used to map rest of the tasks.

The initial tasks can be mapped in two different ways. In the first method, the initial tasks can be mapped on the first free position found in the network that can support the tasks. This may cause the initial tasks to be placed very close to each other. Therefore, when rest of the tasks of different applications are requested to be mapped, the applications need to share the same NoC region, resulting in longer waiting time for a resource to become free for mapping the tasks. This also increases the channel congestion as all the applications are tried to be mapped within a small region. In the second method, virtual clusters are found by partitioning the NoC into regions as shown in Figure 3.3. The clusters are evenly distributed over the NoC to facilitate for uniform utilization of the PEs. The size of each cluster (in terms of number of PEs) is estimated in proportion to the size of the application (in terms of number of tasks) to be mapped in the cluster. One initial task is placed into each virtual cluster in order to map the initial tasks in a distributed manner. This method reduces the interference between different applications and facilitates in the mapping of remaining tasks for each application close to each other, resulting in reduced communication overhead. The cluster boundaries are virtual and hence a common region can be shared by tasks of different applications. This work considers the second method, i.e., the clustering approach.

The Manager Processor (M) knows only the initial tasks. It does not know the whole application graphs. When initial tasks start their execution, communication requests are

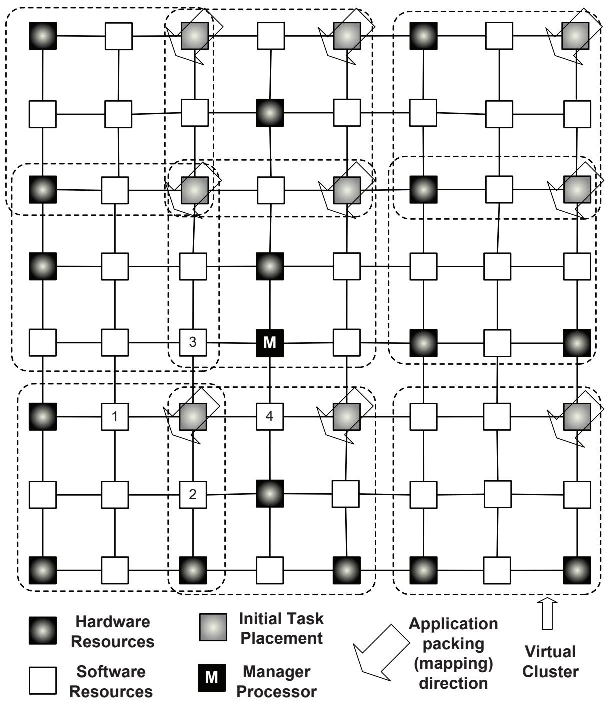


Figure 3.3: Initial tasks placement for mapping (packing) applications

sent to the M to map the slave tasks at run-time. Efficient strategies are required to map the requested slave tasks. Next, we present our packing strategy to accomplish the job of mapping.

3.2.3 Packing Strategy for Minimizing Communication Costs

Our packing strategy attempts to map all the tasks of an application close to each other within a particular region referred to as virtual clusters. The initial task (starting task) of each application is mapped at top-right position within the virtual clusters in a distributed manner using the clustering approach defined above, as shown in Figure 3.3. The packing strategy attempts to map a requested task on the PEs which are around the PE making the request. The PEs are searched in sequence of left, down, top and right denoted as 1, 2, 3 and 4 respectively in Figure 3.3. This way, first, left and down side PEs are searched to find the placement. Now, if neither left nor down side PE is able to execute the task, only then task is tried to be mapped on the top or right side PE according to the above defined sequence. The same strategy is repeated from lower to higher hop distances until a free supported PE is found. Each application follows above defined strategy to map the requested tasks on the MPSoC platform resources.

In case of multiple task nodes communicating with the initial task, the communicating task nodes are requested to be mapped in the order of their assigned task identifier number. For example, if initial task identifier is 0 and its two communicating tasks' identifiers are 1 and 2, then first the task with identifier number 1 gets requested. By requesting the communicating tasks from higher identifier number to lower identifier number (i.e., first 2, then 1) might affect the performance slightly depending upon the number of connected slave tasks to the requested tasks and communication overhead with the slave tasks. In order to achieve maximum performance, the identifiers of tasks are assigned at design-time to optimize performance, based on the communication overhead and connections (edges) between the tasks.

The packing strategy tries to pack (map) each application within a particular virtual cluster with initial task positions as specified above. The strategy tries to map the communicating tasks of an application close to each other within a virtual cluster in a compact manner in order to reduce the communication overhead between the communicating tasks.

3.3 Run-time Mapping Heuristics

In this section, we present run-time mapping heuristics that are motivated by the packing strategy discussed in the previous section. The run-time mapping heuristics are used to find the placement of new requested tasks. These heuristics are light-weight in terms of execution cycles, channel load and packet latency as the heuristics reduce the communication overhead on which all the performance metrics are highly dependent.

Algorithm 1: Packing-based Nearest Neighbor (PNN)

```
Input: $TG(T,E)$, $AG(P,V)$ // task $t_i \in T$ ; PE $p_i \in P$
Output: $mpg$ (mapping $TG(T,E) \rightarrow AG(P,V)$)
  // $NFR[type]$: number of free resource(s) of type $type$ in NoC
 1: Map the initial task ($INI \in T$) at right-top position in a cluster (Figure 3.3);
 2: for all unmapped task $t_i \in T$ that is requested do
 3:   if $NFR[ti_{type}]$ != 0 then
 4:     for all hop_distance = 1 to NoC_limit do
 5:       PE_list = get_packing_ordered_list(hop_distance);
 6:       for all PEs $\in$ PE_list do
 7:         if $p_i$ is free AND $ti_{type}$==$pi_{type}$ then
 8:           Map $t_i$ onto PE $p_i$ and exit to step 17;
 9:         end if
10:       end for
11:     end for
12:   else
13:     insert($t_i$ to Queue($ti_{type}$));
14:     wait until $NFR[ti_{type}]$ != 0;// updated at run-time
15:     release($t_i$ from Queue($ti_{type}$));
16:     Map $t_i$ onto the freed node $p_i$;
17:     insert($p_i$ to $mpg$); update(resources by $mpg$);
18:     wait and goto step 3 if new task $t_i \in T$ is requested;
19:   end if
20: end for
```

3.3.1 Packing-based Nearest Neighbor

This algorithm is based on the packing strategy along with the search space of Nearest Neighbor (*NN*) heuristic proposed in [129], where the search space goes from lower to higher hop distances. The algorithm is presented in Algorithm 1. In order to map a new requested task, the number of free supported resources in the platform is found. If any supported resource is available (step 3) then mapping for the requested task is found as follows. First, resources (PEs) at hop distance of one (step 4) are selected and evaluated to map the task. If none of the PE can support the task then PEs at higher hop distances are selected and evaluated until the mapping is found. The search space to select the PEs goes up to the max_hop_distance (NoC_limit). The selection at each hop distance is done by function ***get_packing_ordered_list****(hop_distance)* (step 5), where PEs are selected according to the packing strategy, i.e. in left, down, top and right order. As soon as, a free supported PE is found, the task is mapped onto the PE, and selection and evaluation process is stopped (step 8). If there is no free supported PE in the platform for the requested task then the task is entered into its corresponding queue (step 13) and waits until a supported PE becomes free (step 14) by finishing execution of some previously mapped task. The queued task is mapped onto the freed supported PE as and when the PE is available (step 16). After mapping the requested task, it is entered onto the mapped list (*mpg*) and resources are updated to have their correct status for next requested task. The same strategy is repeated for each requested task until all the tasks of the application are mapped.

To map multiple applications onto the MPSoC platform, the above described algorithm (PNN) is applied for each application. First, initial tasks of applications are mapped in a distributed manner by the clustering approach as in Figure 3.3. Then, new requested tasks from each application are mapped dynamically, by applying Algorithm PNN. The PNN algorithm reduces the communication overhead by mapping the communicating tasks close to each other in a systematic manner. Therefore, providing improved performance.

3.3.2 Packing-based Best Neighbor

This algorithm is a combination of path load computation approach and the PNN algorithm. The algorithm is presented in Algorithm 2. For each mapping z, the path load (step 9) is computed by Equation Eq. 3.1, where $r_{ch(i,j)}$ and $r_{ch(j,i)}$ are the rates in the individual channels, from the master to the new requested slave and the rates in the channels in opposite direction, respectively.

$$cost_z = \sum r_{ch(i,j)} + \sum r_{ch(j,i)} \qquad \text{(Eq. 3.1)}$$

Algorithm 2: Packing-based Best Neighbor (PBN)

Input: *TG(T,E)*, *AG(P,V)* // task $t_i \in T$; PE $p_i \in P$ (PE)

Output: *mpg* (mapping *TG(T,E)* $\rightarrow$ *AG(P,V)*)

```
 1: Map the initial task ($INI \in T$) at right-top position in a cluster (Figure 3.3);
 2: for all unmapped task $t_i \in T$ that is requested do
 3:   if $NFR[ti_{type}]$ != 0 then
 4:     for all hop_distance = 1 to NoC_limit do
 5:       weight = MAX_VALUE; // some large value
 6:       PE_list = get_packing_ordered_list(hop_distance);
 7:       for all PEs ∈ PE_list do
 8:         if $p_i$ is free AND $ti_{type}$==$pi_{type}$ then
 9:           weightTemp = calcChannelLoad(when $t_i$ mapped onto $p_i$);
10:           if weightTemp < weight then
11:             weight = weightTemp;
12:             Select node $p_i$ temporarily to map $t_i$;
13:           end if
14:         end if
15:       end for
16:       if weight < MAX_VALUE then
17:         Map $t_i$ onto PE $p_i$ and exit to step 22;
18:       end if
19:     end for
20:   else
21:     Perform steps 13 to 16 from Algorithm PNN;
22:     insert($p_i$ to mpg); update(resources by mpg);
23:     wait and goto step 3 if new task $t_i \in T$ is requested;
24:   end if
25: end for
```

In algorithm PNN, after finding the PE list at each hop distance, the evaluation stops when first free supported PE is found. However, in algorithm PBN, all the free supported PEs are evaluated (selected temporarily- step 12) after finding the PE list and the PE with minimum path load is chosen for final mapping in order to get the best neighbor from the available neighbors. The evaluation process is stopped for higher hop distances if a mapping is found.

As this heuristic includes path load computation, it is a congestion aware mapping heuristic that tries to distribute the channel load in the NoC. Thus, in addition to mapping the tasks in close proximity to reduce the communication overhead, this heuristic also tries to distribute load in the channels more uniformly, resulting in reduced average channel load. Average packet latency also gets reduced as it depends on distance between source and destination PE and congestion in the communication path which gets reduced by considering congestion in channels during path load computation.

3.3.3 Packing-based Time-bounded Best Neighbor

The PBN heuristic mentioned earlier takes additional time for performing path load computations over the PNN, resulting in increased overall execution time. The additional time in PBN can get compensated with possible reduction in communication time that might get minimized due to communication overhead reduction by distributing the channel loads more uniformly. This requires suitable scenarios to be evaluated. The path load computation time can be reduced in all the scenarios if we allow the load computation for some particular time instead of allowing it until the best PE is found.

The Packing-based Time-bounded Best Neighbor (PTBN) algorithm incorporates time bounded evaluation of the neighbors and is presented in Algorithm 3. The time bounded consideration discards evaluation of all the neighbors for their path load in order to reduce the overall execution time. The time bound starts from the point where PNN converges (step 11, Algorithm 3). The algorithm evaluates neighbors for a particular time (TIME_BOUND) towards getting a better neighbor, which is then selected for final mapping. Therefore, the algorithm tries to reduce overall execution time. For a large value of time bound, this algorithm will behave like PBN and like PNN for a very small value of the time bound. The algorithm is analyzed for varying values of time bound and its behavior on different performance metrics has been discussed in Section 3.4.

Algorithm 3: Packing-based Time-bounded Best Neighbor (PTBN)

```
Input: TG(T,E), AG(P,V), TIME_BOUND // task $t_i \in T$ ; PE $p_i \in P$ (PE)
Output: mpg (mapping $TG(T,E) \rightarrow AG(P,V)$ )
 1: Map the initial task ($INI \in T$) at right-top position in a cluster (Figure 3.3);
 2: for all unmapped task $t_i \in T$ that is requested do
 3:   if $NFR[ti_{type}]$ != 0 then
 4:     for all hop_distance = 1 to NoC_limit do
 5:       StartTime = 0; PEevaluated = 0; weight = MAX_VALUE;
 6:       PE_list = get_packing_ordered_list(hop_distance);
 7:       for all PEs ∈ PE_list do
 8:         if $p_i$ is free AND $ti_{type}$==$pi_{type}$ then
 9:           PEevaluated++;
10:           if PEevaluated == 1 then
11:             StartTime = CurrentTime(); // time when PNN will converge
12:           end if
13:           weightTemp = calcChannelLoad(when $t_i$ mapped onto $p_i$);
14:           if weightTemp < weight then
15:             weight = weightTemp;
16:             Select node $p_i$ temporarily to map $t_i$;
17:           end if
18:           if CurrentTime() - StartTime > TIME_BOUND then
19:             Go to step 23;
20:           end if
21:         end if
22:       end for
23:       if weight < MAX_VALUE then
24:         Map $t_i$ onto PE $p_i$ and exit to step 29;
25:       end if
26:     end for
27:   else
28:     Perform steps 13 to 16 from Algorithm PNN;
29:     insert($p_i$ to mpg); update(resources by mpg);
30:     wait and goto step 3 if new task $t_i \in T$ is requested;
31:   end if
32: end for
```

3.4 Performance Evaluation

Experiments are performed by *co-simulation* in *ModelSim* (*System-C* for applications and *RTL-VHDL* for the NoC). Evaluated performance metrics are *total execution time*, *average channel load* and *average packet latency* for applications.

The simulation platform used for our experiments is similar to that in [129]. The processing elements (PEs) are modeled using *System-C*. Two different *System-C* threads are used to model the PEs, one for the Manager Processor (M) and another for rest of the PEs as *Mthread* and *TASKthread*, respectively. The *Mthread* is responsible for the MPSoC *resource management*, *task mapping*, *task scheduling* and *task configuration*. This thread contains channels occupation metrics, PEs occupation metrics and scheduling queues to manage system use. The resource metrics are updated at run-time by monitoring the resources status with the help of monitors attached to all the NoC ports. The *TASKthread* is responsible for the task behavior implementation that is described by a configuration file. This file contains execution time and communication rates, which can be customized.

Each application is modeled as in Figure 3.2, with an initial task, hardware tasks and software tasks. The values present on the edges represent the volume and rate of data to be exchanged between the master and slave as explained in *application task graph* definition of Section 3.2.1. Each task transmits from 200 to 500 packets (data volumes (V) on the edges as in Figure 3.2) with size varying from 100 to 400 16-bit flits. The packet processing time is fixed. Hardware and software tasks allocation time is taken as 1300 and 100 clock cycles respectively [154]. Initial tasks are mapped onto the processors, so the configuration time is the same as that of software tasks. The simulation is performed at varying time bound to find the optimal point providing better results for all the considered performance metrics.

The experiments are performed for different scenarios. In each scenario, 20 identical tree like applications (parallel benchmarks have this profile) are taken with varying injection rate (% usage of available channel bandwidth). The applications are generated using *Task Graph For Free (TGFF)* tool [92]. The results are shown for following simulation scenarios:

(i) Each application having 4 tasks (1 initial, 2 software and 1 hardware task).

(ii) Each application having 7 tasks (1 initial, 4 software and 2 hardware tasks).

(iii) Each application having 10 tasks (1 initial, 6 software and 3 hardware tasks).

(iv) Each application having 20 tasks (1 initial, 13 software and 6 hardware tasks).

(v) Each application having 30 tasks (1 initial, 20 software and 9 hardware tasks).

The NoC is modeled in VHDL [64], in an 8×8 2D-mesh topology. NoC is responsible for data transfer between the tasks mapped on different PEs. As handshake protocol is used to transfer the data, each flit is transmitted in two clock cycles, thereby limiting the available channel bandwidth to 50% of its capacity. In the NoC (Figure 3.3), one PE is used as manager processor (M), 16 as hardware resources and 47 as software resources. Figure 3.3 shows placement of initial tasks within the clusters for executing nine applications at a time. Each cluster corresponds to an independent application. As clusters are virtual, an application can occupy resources of other clusters. The number of initial task supported nodes determines the maximum number of simultaneously running applications and the number is changed according to the number of tasks in each application for the considered scenario. For applications containing large number of tasks, the number of initial task supported nodes are reduced so that the tasks queuing overhead in case of no free supported resources can be kept almost the same for all the scenarios. This consideration avoids extra queuing overhead for applications containing large number of tasks and facilitates for lesser overall execution time.

Results obtained from our proposed heuristics PNN, PBN and PTBN are compared with the state-of-the-art run-time mapping heuristics Path Load (PL), Nearest Neighbor (NN) and Best Neighbor (BN) proposed in [129]. The PL, NN and BN also try to reduce communication bottlenecks but do not perform well for all the scenarios.

3.4.1 Total Execution Time

The total execution time is the time taken to finish the execution of all the applications to be mapped on the platform. It comprises of mapping time (time to find the placement in the 8×8 2D-mesh), configuration time, communication time, waiting time (when no free resource in the platform) and computation (processing) time, amongst which communication time dominates. The computation of a packet corresponding to a task mapped on a PE starts just after it has been received and the computation is finished before receiving the next packet for the same task. So, if the computation time is less than the time interval between receiving two consecutive packets corresponding to the same task on the PE, then the computation time will get absorbed within the communication time. At this stage of our work, the computation time is taken small enough so that it does not contribute much to the total execution time, i.e. communication overhead dominates computation overhead.

Our proposed mapping heuristics map the communicating tasks in close proximity, resulting in reduced communication overhead and generated traffic in channels (channel congestion). The communication overhead and generated traffic are more when the communicating tasks are mapped on distant apart PEs. The reduced communication overhead facilitates for faster communication, i.e. reduced communication time and thus the reduced total execution time.

Figure 3.4 shows average execution time for three simulation scenarios at varying injection rates (% usage of available bandwidth) when heuristics PL, NN, PNN, PBN and PTBN are employed. For each task, time bound for PTBN has been considered as quotient of (the difference of the time taken by PBN and PNN heuristics in order to find a mapping for the task)/2, which is like allowing for about half of the time to PTBN as compared to PBN for exploring the best mapping. The following observations can be made from the Figure 3.4. First, the execution time decreases with increase

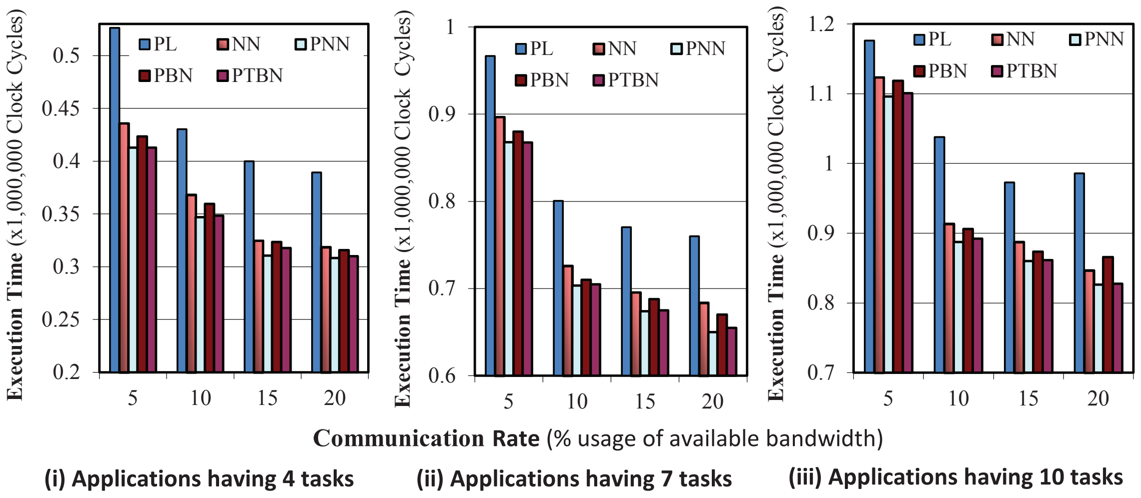


Figure 3.4: Total execution time for PL, NN, PNN, PBN and PTBN heuristics for three simulation scenarios

in communication rate as communication time is reduced by using more percentage of available bandwidth that provides faster communication. Second, execution time by PNN is reduced over the PL and NN. Third, execution time by PBN is more when compared to PNN due to path load computation overheads. Fourth, PTBN shows execution time close to that of the PNN as it performs time bounded path load computations. It has been observed that the difference in execution time by PBN and PNN increases when applications containing larger number of tasks are considered. This happens because path load computation overhead in PBN increases with the number of tasks.

The complexities of the heuristics have been computed in terms of number of PEs that can be identified to map a task. In order to map the task, the worst-case requires identification of all the PEs of the NoC in order to map the task. Therefore, the complexity has a linear relationship to the number of PEs. Our analysis in time complexity shows that all the heuristics have time complexity of the same level and is of the order of $O(C)$, where C is the number of PEs in the NoC. All the heuristics execute almost in similar time with minimal differences.

3.4.2 Average Channel Load

The average channel load represents the NoC use. It is calculated by looking the loads in all the channels at a fixed clock cycle interval until all the applications finish their execution. The load in the channels depends upon the traffic produced by the communicating tasks while communicating from different PEs. The traffic produced can be reduced if the communication overhead between the tasks is lowered by mapping them close to each other. Additionally, it has to be noted that the reduced traffic decreases the communication overhead of other communicating tasks mapped on different PEs.

In our mapping heuristics, first heuristic (PNN) does not consider traffic during mapping, but explores the proximity of communicating tasks. In the second heuristic (PBN), we consider the traffic during mapping, thus trying to distribute the channel load more uniformly leading to reduced average channel load. However, the heuristic involves significant traffic consideration overhead. The third heuristic (PTBN) reduces traffic consideration overhead while trying to distribute the channel load. Our heuristics map tasks for each application in a systematic manner within a particular cluster, thereby reducing the chances of interference for the used channels by different applications. This reduces the chance of having high load in channels. Thus, average channel load is significantly reduced when proposed heuristics are employed.

Figure 3.5 plots average channel load for three simulation scenarios at varying injection rates (% usage of available bandwidth) when heuristics BN, PBN and PTBN are employed. We have not shown channel load when PNN is employed as it always shows worst channel load than PBN. A couple of observations can be made from the Figure 3.5. First, the average channel load increases with increased communication rate as more traffic gets generated in the channels. Second, average channel load by PBN is less when compared to BN and PTBN as it gets chance for exploring the best neighbor till it has not been found. Third, PTBN shows the average channel load close to that of the PBN

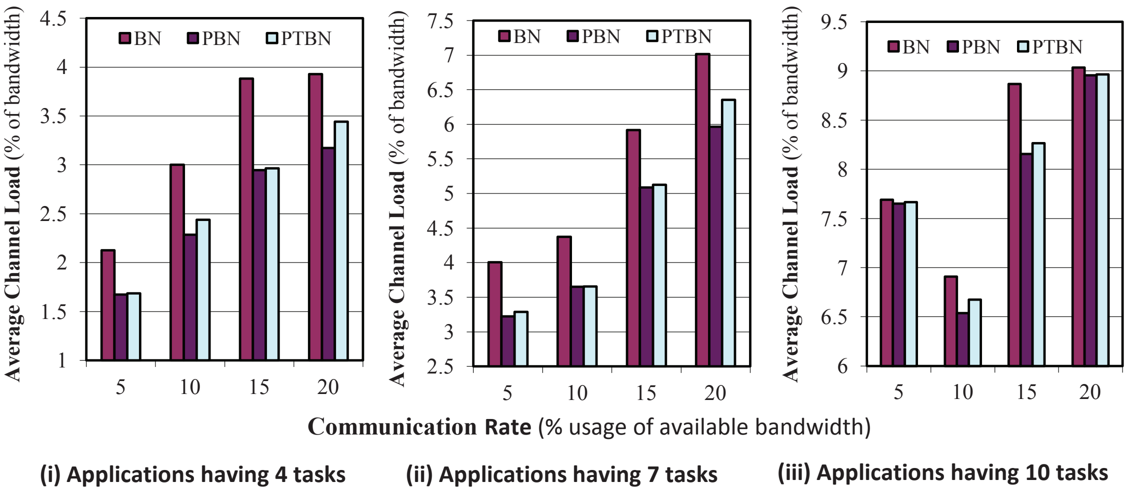


Figure 3.5: Average channel load for BN, PBN and PTBN heuristics for three simulation scenarios

as it finds the same best neighbor as of the PBN for most of the tasks in the allowed time bounds. In the first scenario (Scenario i), PBN and PTBN reduces the channel load by 3.74% and 2.87%, by 15.8% and 13.55% in the second scenario (Scenario ii) and by 22% and 18.58% in the third scenario (Scenario iii) when compared to BN. It can be seen that the improvement by PBN and PTBN does not differ much so it would be better to employ PTBN for performing mapping as it reduces total execution time as well. The channel load reduction trend shows that when number of tasks in each application is less, our heuristic performs better because of better packing of the tasks.

3.4.3 Average Packet Latency

The average packet latency depends on *1)* the distance between the source and destination PEs on which communicating tasks are mapped and *2)* the congestion in the communication path. It is measured as the average time each packet takes in traversing from source to destination PE. As described earlier, the proposed mapping heuristics map the communicating tasks close to each other and thereby reduce NoC congestion as well. The congestion in the communication path is reduced more by PBN and PTBN as

Scenarios	Rates	**Avereage Packet Latency** (Clock Cycles)		
		BN	**PBN**	**PTBN**
Applications having 10 tasks	**5%**	144	139	141
	10%	233	224	227
	15%	344	332	338
	20%	438	423	428

Table 3.1: Average Packet Latency Measured in Clock Cycles

they consider congestion in channels during mapping. Thus, PBN and PTBN provide reduced average packet latency over BN.

Table 3.1 presents the latency results for a simulation scenario at varying injection rates (% usage of available bandwidth) when different heuristics are employed. Network congestion depends directly on the communication rate and thus the average packet latency, as shown in Table 3.1. It can be seen that average packet latency for our heuristics PBN and PTBN get reduced when compared to BN. Similar results are obtained for other simulation scenarios.

3.4.4 Effect of Time Bound

The effect of different time bound values has been analyzed on execution time in order to find a time bound value that should lead to better results for all the performance metrics when heuristic PTBN is employed. It is evident that the heuristic will find a better neighbor as we allow for more time bound to evaluate the neighbors. Average channel load and packet latency decrease with the time bound as selecting the better neighbors for mapping helps in homogeneous distribution of channel loads (congestions). The reduced congestion helps in faster communication and thus in reducing the total execution time. However, the allowed time bound gets added to the total execution time, so we may not get optimized execution time at all the time bound values.

In order to perform the time bounded analysis of PTBN heuristic, first, mapping time for each task is captured by employing PNN and PBN. The time bound provided

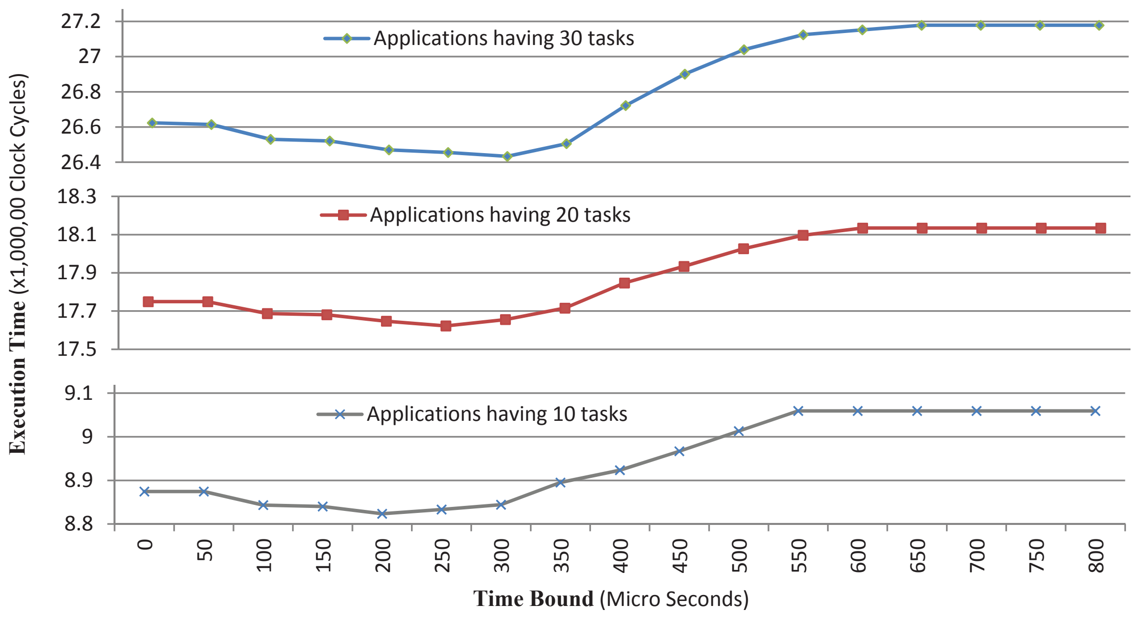


Figure 3.6: Execution time at varying time bound values

to PTBN starts from the time taken by PNN to find a mapping. The difference in mapping time by PBN and PNN provides the maximum value of time bound that should be provided to PTBN in order to have its behavior like PBN. Providing a time bound value of more than the difference value would not affect the performance further and the algorithm will behave like the PBN.

Figure 3.6 shows total execution time for three simulation scenarios when heuristic PTBN is employed at different values of time bound. The execution time shown is for the injection rate of 10% (% usage of available bandwidth). Similar behavior is obtained at other injection rates too. A couple of observations can be made from Figure 3.6. First, scenarios containing large size applications (applications containing larger number of tasks) show higher execution time at all the time bound values as more number of tasks needs to be executed. Second, execution time for all the scenarios first falls for some initial values of time bound and then starts increasing after reaching a minimum and finally becomes constant after some value of time bound. The initial falling trend is obtained due to more reduction in communication time than the allowed time bound, contributing to the total execution time. Communication time gets reduced as better neighbors are selected which provide lower congestion and thus faster communication. The falling trend continues till the reduction in communication time is greater than the allowed time bound. Thereafter, the increasing trend is obtained as allowed time bound adds more to the total execution time than the reduction in the communication time. After some values of time bound, execution time does not get affected as the heuristic terminates after evaluating all the neighbors even before the allowed time bound. It can be observed that for applications with larger number of tasks, the minimum execution time and start of constant execution time region is obtained at higher values of time bound. This is because the heuristic needs to evaluate larger number of neighbors to get better neighbors for the tasks, which provide more reduction in communication time than the allowed time bound. It can also be observed that the difference in maximum and minimum execution time increases with the number of tasks in considered applications. Therefore, all the performance metrics can be optimized simultaneously when employing PTBN for a particular time bound.

3.5 Summary

In this chapter, we have proposed a packing strategy that aims to map communicating tasks within a virtual cluster in order to reduce the communication overhead. Three efficient run-time mapping heuristics based on the packing strategy have been presented. The heuristics target NoC-based heterogeneous MPSoCs, where each PE is assumed to support a single task. The PE can be either a software PE (Instruction Set Processor) or a hardware PE (Reconfigurable Hardware).

The first heuristic tries to map the tasks of an application in close proximity, thereby reducing the communication overhead of the communicating tasks. For each task to be mapped, the heuristic chooses the first available supported PE that is found with respect to its master task PE. While this approach is fast, it does not consider traffic during mapping. The second heuristic evaluates all the free supported neighboring PEs and selects the best neighboring PE for the mapping in order to reduce the traffic. As this heuristic considers traffic in addition to the proximity of tasks, it results in more uniform distribution of channel loads but at the cost of increased evaluation overhead. In particular, the evaluation overhead for determining the best neighbor increases with the number of tasks in the applications to be mapped. The third heuristic overcomes the overhead limitation by restricting the number of evaluations by setting an evaluation time bound. Detailed analysis show that the evaluation time bound facilitates the identification of better neighbors for each task such that all the performance metrics are optimized. This has led to a marginal decrease in the overall execution time when compared to the second heuristic.

The proposed heuristics have been evaluated using an 8×8 NoC-based MPSoC platform for different application scenarios. We clearly demonstrate that the proposed heuristics can consistently result in notable reduction in the communication overhead. In addition, we have investigated different scenarios and evaluated performance metrics of interest such as total execution time, average channel load and average packet latency. Experimental results show that reduction in the average channel load can be up to 22% when compared to a state-of-the-art mapping heuristic. The total execution time and average packet latency are also reduced. While the proposed techniques in this chapter have led to significant gains, they are applicable for MPSoCs that support a single task on each PE. In the next chapter, we will present mapping heuristics for MPSoCs with multi-task supported PEs in order to demonstrate the applicability and efficiency of our methods in more realistic scenarios.

Chapter 4

Mapping Multiple-Tasks-per-Processing-Element

In the previous chapter (Chapter 3), we presented three run-time mapping techniques for NoC-based heterogeneous MPSoCs. The techniques were shown to outperform state-of-the-art techniques. However, each processing element (PE) of MPSoCs was able to support only a single task. Supporting a single task on each PE is not a realistic scenario. Further, we cannot exploit the possible advantages by the existing techniques when multi-task supported PEs are considered. Therefore, the techniques need to be extended to MPSoCs where more than one task can be mapped on each PE.

In this chapter, we extend the mapping techniques proposed in Chapter 3 to MPSoCs containing multi-task supported PEs. The extended techniques take the advantage of multi-task supported PEs by allowing the mapping of communicating (adjacent) tasks on the same PE whenever possible, which results in reduced communication overhead. We validate our extended techniques with different type of application sets and demonstrate that the extended techniques show significant performance improvement when compared to the existing techniques. Part of the work in this chapter has been published in [C-2], [C-3] and [J-1].

The rest of the chapter is organized as follows. In Section 4.1, we introduce the extended MPSoC architecture containing multi-task supported PEs and analyze the multi-task mapping on the MPSoC PEs. Section 4.2 introduces the packing strategy applied to the extended architecture. The extended run-time mapping techniques based on the packing strategy are presented in Section 4.3. We present our experimental results in Section 4.4 and summarize the chapter in Section 4.5.

4.1 Target MPSoC Architecture

The MPSoC architecture used here is an extended version of that used in Chapter 3. In Chapter 3, each processing element (PE) was capable of supporting only a single task. In the extended version, the PEs are modeled to support more than one task. The maximum number of tasks to be supported on the PEs depends upon the available memory space and reconfigurable area for the software and hardware tasks respectively.

In an MPSoC where each processor supports a single task, there is no sharing of the processor by multiple tasks. Thus, the task assigned to the stand-alone processor is executed without interruptions as it is the sole master of all available processing capability of the processor. However, in case of a multi-task supported processor, tasks assigned to the processor compete with each other for acquiring the processor attention and end-up preempting each other as a result of mutual competition. The assigned tasks can manage their execution by switching among them after completing one operation of a task, similar to the execution in OSs by serving the tasks in time multiplexed manner. Tasks mapped on the same processor can communicate through some common register or memory space if required. For supporting multiple tasks on hardware PEs, reconfigurable areas (RAs) used as the hardware PEs can be considered large enough to support multiple configurations for multiple tasks in parallel.

Incorporating multi-task supported PEs in the platform can lead to performance improvement. The main advantage of the PEs can be taken by mapping the communicating tasks on the same PE that will result in reduced communication overhead. The communicating tasks mapped on the same PE can interact with each other very fast as they do not require any network resources. The reduced communication overhead minimizes the time and energy required in communication. Thus, the performance metrics depending upon the communication overhead are optimized.

The work in this chapter too focuses on *task mapping*, *task scheduling*, *resource control* and *configuration control* similar to Chapter 3.

4.2 Supporting Multiple-Tasks-per-Processing-Element

In this section, we describe the packing strategy for the extended MPSoC architecture. The definitions of the *application task graph*, *MPSoC architecture* and *mapping* are the same as described in Chapter 3. In this chapter, the PEs of the MPSoC are capable of supporting multiple tasks, so the task set p_{tasks} of a tile in the MPSoC can contain more than one task and the maximum number of tasks on the tile can go up to its capacity p_{cap}. The capacity of a tile is determined by the available memory space or reconfigurable area to configure the tasks on the tile.

4.2.1 Placing Initial Tasks

Before applying the packing strategy, the initial tasks (defined in *application task graph* definition of Chapter 3) of the applications should already be mapped in the MPSoC architecture. After the initial tasks are mapped, the remaining tasks of each application are mapped by applying the packing strategy at run-time.

Similar to Chapter 3, here also, a clustering approach has been adopted to find the placement of initial tasks. They are mapped as far away as possible while avoiding the edges in a distributed manner, as shown in Figure 4.1. The distributed mapping of the initial tasks reduces the interference between different applications and helps in mapping

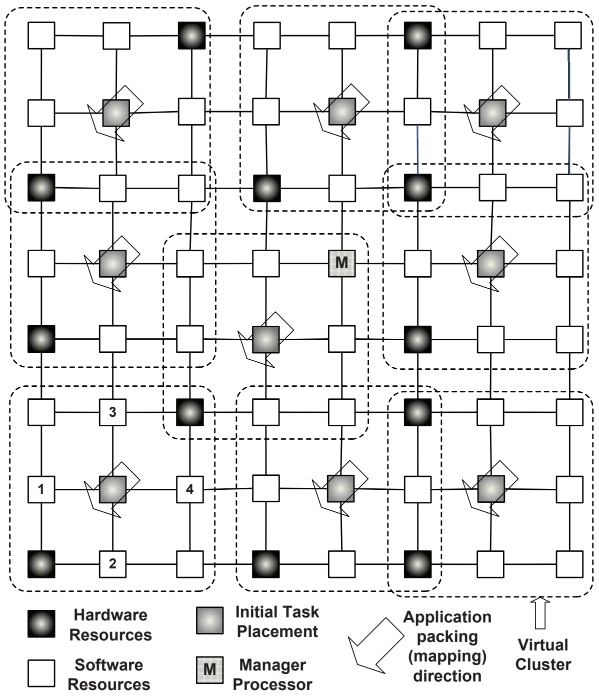


Figure 4.1: Initial tasks placement for mapping applications

of the new communicating tasks for each application close to each other in order to reduce communication overhead.

After the initial tasks are mapped and start their execution, communication requests are sent to the manager processor (M) to map the slave tasks at run-time. The requested tasks can be mapped by the packing strategy described subsequently.

4.2.2 Packing Strategy to Support Multiple-Tasks-per-Processing-Element

To map a requested task, firstly, the task is tried to be mapped at the same node (master task PE) making the request as the processing resources can support more than one task. If the task is not supported by the node making the request then it is tried to be mapped on the PEs around the node making the request at hop distance of one. The PEs are searched in sequence of left, down, top and right denoted as 1, 2, 3 and 4 respectively in Figure 4.1 for one application in the most bottom-left cluster. The same strategy is followed from lower to higher hop distances until a free supported PE is found. Each application follows the similar strategy to map the requested tasks on the MPSoC resources.

With this strategy, each application tries to map its requested task towards bottom-left (either on down or left PE) side within the cluster, hence the PEs present on top-right edge of the cluster may be used by tasks of other applications that are also trying to map their tasks towards the bottom-left. In this manner if one application is getting mapped then the applications that are tried to be mapped on top-side, right-side or top-right side may get the free resources on the top and right edges of the first application's cluster and tasks of the other applications can be mapped on these resources. This strategy is applied to all the applications to be mapped and most of the applications get the free resources from other application's top-right edge of the cluster. Thus, resource utilization is increased. Additionally, as each platform resource can support more than one task, communicating tasks get mapped on the same resource, resulting in further reduction in communication overhead and making the mapping more compact.

4.3 Run-time Mapping Heuristics

This section describes the run-time mapping heuristics that are motivated by the packing strategy extended for the multi-task per PE mapping. The heuristics are used to find the best placement for new requested tasks.

4.3.1 Packing-based Nearest Neighbor

This algorithm is an extension of the algorithm PNN proposed in the previous chapter. The extended algorithm assumes multi-task supported PEs in the MPSoC platform. In order to map a requested task, the extended algorithm first selects and evaluates resource (PE) at the requesting node position (at zero hop distance; step 4 of Algorithm PNN) to map the task unlike evaluating the resources at one hop distance as in the previous chapter. If the requesting node PE can't support the task then PEs at higher hop distances are selected and evaluated until the mapping is found. The search space to select the PEs goes up to the NoC limit (max_hop_distance) and the selection at each hop distance is done by function ***get_packing_ordered_list***(hop_distance) as in PNN described in the previous chapter. The selection and evaluation process is stopped as soon as a free supported PE is found to map the task (step 8). The same strategy is followed by each requested task of the application to be mapped. In order to map multiple applications, the algorithm PNN is applied for each of the application after mapping the initial tasks in a distributed manner by the clustering approach as in Figure 4.1.

4.3.2 Packing-based Best Neighbor

This algorithm is an extension of the PBN algorithm proposed in the previous chapter. The extended algorithm facilitates mapping of multiple tasks on the same PE. Therefore, the PEs are searched from the requesting node position to the NoC_limit (from hop_distance = 0 to NoC_limit; step 4 of PBN in the previous chapter). At each hop_distance, all the free supported PEs are evaluated for their imposed path load, whereas the PNN stops evaluation as soon as a free supported PE is found. The path load (PL) is computed from Equation Eq. 3.1 of Chapter 3 and the PE imposing minimum PL is chosen for final mapping. The free supported PE at zero hop distance has zero path load as no channel is involved.

This heuristic reduces average channel load and packet latency when compared to PNN as congestion in channels get reduced by considering the traffic in channels during mapping. As mentioned in the previous chapter, total execution time may get increased as compared to PNN due to additional time taken in path load computation for considering the traffic.

4.3.3 Packing-based Time-bounded Best Neighbor

This algorithm incorporates time bounded path load computations for evaluating neighboring PEs so that total execution time can be reduced along with the average channel load and packet latency. The PTBN algorithm proposed in the previous chapter is extended to support multiple tasks on each PE. Time bounded computation tries to reduce overall execution time and also tries to find a better neighbor in order to distribute the channel loads more homogeneously. The algorithm behaves like PNN for a small value of time bound and like PBN for large values of time bound.

4.4 Performance Evaluation

In this section, we present our results for the extended mapping techniques targeting MPSoC platforms containing multi-task supported PEs. The evaluated performance metrics are *total execution time*, *average channel load* and *average packet latency.*

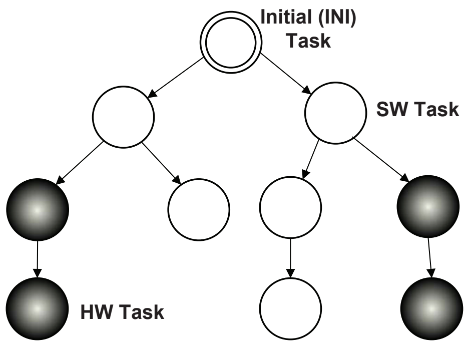
The simulation platform used for experiments is an extended version of that used in Chapter 3. The platform was first extended such that multiple tasks can be mapped onto each hardware resource (PE) by incorporating large reconfigurable area (RA). Later, it was extended to support multiple tasks by each processing node (e.g., ISPs & RAs). However, it is known that larger memory space and reconfigurable area will be required to support more number of software and hardware tasks respectively. The memory space required at each node depends on both the number of tasks and size of the tasks to be mapped. All the tasks mapped on a node need to be considered because while one task is running others need to be stored (kept active) in memory. The tasks mapped on a node get executed one after another in time multiplexed manner. For the experimentation, all the tasks are considered of the same size so the memory space is governed by the number of tasks (all same size). We have performed the experiments by varying the number of tasks per node.

For tasks of different sizes, the mapping techniques need to check for available memory space at a PE while trying to map a task on the PE. If required memory space is available on the PE, the task gets mapped; otherwise the task is mapped on a different PE satisfying the memory requirement. After mapping the task on PE, the remaining capacity of the resource (PE) in terms of available memory space is updated to have accurate information about the resource availability for the following task mappings.

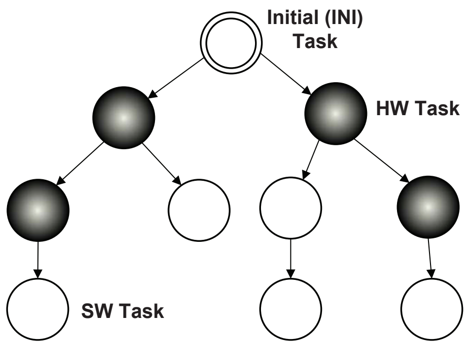
The optimal number of tasks per PE will depend upon the imposed computation and communication overhead on the PEs while supporting multiple tasks. The application structure governs the computation and communication overhead and thus the number of optimal tasks per PE. For finding optimal number of tasks per PE, the strategy can be executed repeatedly by considering different number of tasks per PE. Thereafter, based on the execution providing maximum performance, the optimal number of tasks per PEs can be found.

Each application is modeled as in the previous chapter (Figure 3.2 of Chapter 3), with an initial task, hardware tasks and software tasks. Each task transmits from 200 to 500 packets (data volumes) on the edges. After receiving a packet corresponding to some particular task on a PE, the packet is processed for some definite time before starting the processing of next packet corresponding to the same task on the PE. If two tasks are mapped on the receiver PE, then processing time of packets corresponding to each task get doubled as the packets are processed in time multiplexed manner. The processing time gets tripled for packets when processed on a PE containing three tasks and so on. The packet processing time on a PE containing a single task is fixed. The allowed time bound to evaluate the neighbors is varied to analyze the behavior of PTBN heuristic on different performance metrics.

A lot of scenarios have been evaluated to validate our extended mapping techniques. Here, we present results for the following two scenarios:



4.2.a: An application for scenario i.



4.2.b: An application for scenario ii.

Figure 4.2: Applications for the two simulated scenarios

(i) Applications having hardware communicating tasks at leaf (Figure 4.2.a).

(ii) Applications having hardware communicating tasks in between initial and leaf tasks (Figure 4.2.b).

In each scenario, 20 identical applications, each one with 10 tasks (Figure 4.2) are considered. The injection (communication) rate is varied from 5 to 20% of available bandwidth.

An 8×8 NoC-based MPSoC has been considered. In the MPSoC architecture, one node is used as manager processor (M), 44 nodes as software (SW) resources and 19 nodes as reconfigurable areas. The node used for the manager processor is considered to support a single task as it has additional overhead for managing the whole system.

The number of simultaneously running applications (initial tasks) is varied according to the processing capability of the platform that gets increased when number of tasks to be supported at each PE is increased. This variation is required to utilize all the platform resources, otherwise some resources might be just idle and doing nothing. The best results are obtained with 10, 18 and 20 simultaneously running applications (initial tasks) when each platform PE supports 2, 4 and 8 tasks (except for PE used as manager processor) respectively.

Results obtained from our proposed heuristics PNN, PBN and PTBN are compared with the latest approaches Nearest Neighbor (NN) and Best Neighbor (BN) presented in [129]. NN and BN are extended so that multiple tasks can be mapped on the PEs in order to make a fair comparison with our extended heuristics. The NN and BN are executed for the platform where each PE is considered to support a single task and for the extended platform to observe the improvement by incorporating multi-task supported PEs. While executing for the platform supporting Single Task on each PE (ST/PE), NN and BN are referred to as NN(ST/PE) and BN(ST/PE), respectively. As mentioned earlier, the platform was first extended to support Multiple Tasks on each RA (MT/RA). In this case, NN and BN are referred to as NN(MT/RA) and BN(MT/RA), respectively. Finally, the platform was extended to support Multiple Tasks on each PE (MT/PE), and heuristics NN and BN are referred to as NN(MT/PE) and BN(MT/PE), respectively.

4.4.1 Total Execution Time

The total execution time is the time taken to finish the execution of all the applications, which is computed in the similar manner as in Chapter 3. Similar to Chapter 3, here also, computation overhead is negligible as compared to the communication overhead. Therefore, communication time dominates the total execution time. Our mapping heuristics take advantage of the multi-task supported PEs by mapping the communicating tasks on the same PE in order to reduce communication overhead and thus the communication time.

Figure 4.3 shows total execution time for the two simulated scenarios at different communication rate (% usage of available bandwidth) when different heuristics are employed. The shown results are for maximum two tasks per node. For each task, time bound for PTBN has been considered as (the difference of the time taken by PBN and PNN heuristics in order to find a mapping for the task)/2, to allow for a reasonable time

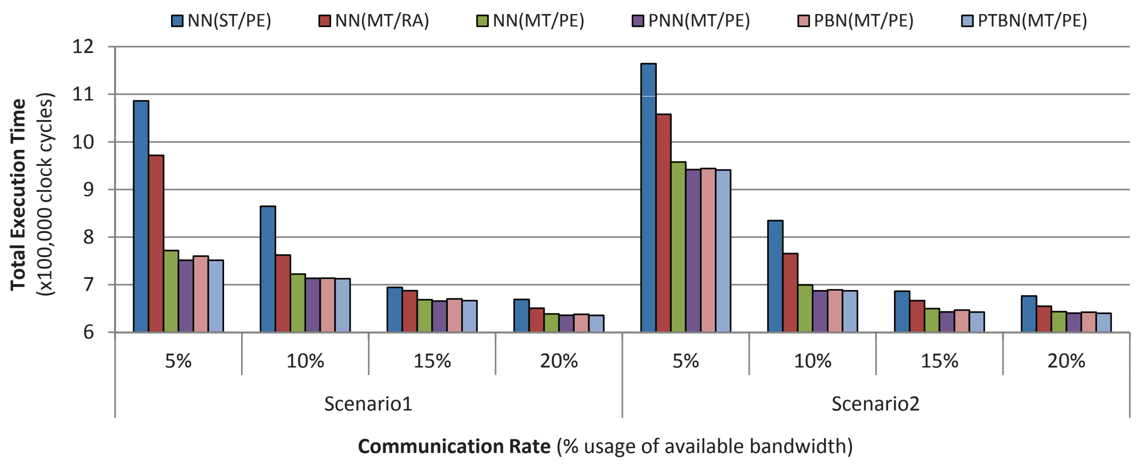


Figure 4.3: Total execution time for two simulated scenarios at different communication rates

for exploring the best mapping. It can be observed from the figure that the execution time is significantly reduced when heuristic NN is employed for the extended platforms. Our heuristics show further reduction. PNN and PTBN show almost the same and minimum execution time. PBN has higher execution time over PNN and PTBN as it explores for the best mapping for a longer time. All the heuristics have similar execution time for the extended final platform due to their similar time complexity that is of $O(C)$, where C is the number of PEs in the NoC.

4.4.2 Average Channel Load

The average channel load is calculated in the same manner as in Chapter 3. Figure 4.4 shows average channel load for the two simulated scenarios at different communication rate (% usage of available bandwidth) when different heuristics are employed to support maximum two tasks on each node. The channel load is not shown for NN and PNN heuristics as they will always provide worst values than the congestion aware heuristics shown in the figure. When extended heuristics are employed, adjacent tasks get mapped on the same node, resulting in reduced communication overhead. Thus, channel load is reduced as it depends directly on communication overhead.

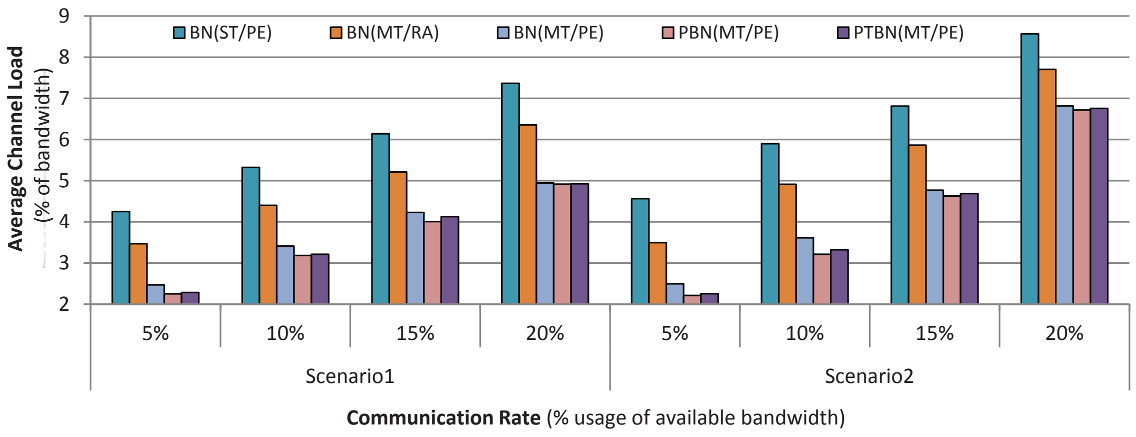


Figure 4.4: Average channel load for two simulated scenarios at different communication rates

A couple of observations can be made from Figure 4.4. First, average channel load is greatly reduced when heuristic BN is employed for the extended platforms allowing mapping of the adjacent (communicating) tasks on the same PE. Second, average channel load by our proposed heuristics PBN and PTBN is reduced when compared to BN. Third, PBN shows minimum average channel load as the heuristic distributes loads in the channels more uniformly after finding the best neighbor for each task, whereas PTBN tries to find a better neighbor within the allowed time bound.

4.4.3 Average Packet Latency

Average packet latency is the average time taken by each packet when traversing from source to destination PE. The time taken depends upon distance between the source and destination PE and congestion in the network. The incorporation of multi-task supported PEs facilitates mapping of the communicating tasks on the same PE, reducing the traffic produced. Therefore, average packet latency for other tasks communicating from different PEs gets reduced. The latency for the packets of the communicating tasks mapped on the same PE is reduced significantly as the packets can be easily exchanged on the PE without needing any channel but we have not considered latency for these packets while

		Average Packet Latency (Clock Cycles)				
	Rates	BN (ST/PE)	BN (MT/RA)	BN (MT/PE)	PBN (MT/PE)	PTBN (MT/PE)
Scenario1	**5%**	134	130	125	124	124
	10%	237	231	221	216	218
	15%	331	329	327	323	324
	20%	436	432	427	421	421

Table 4.1: Average packet latency at different communication rates

calculating the average packet latency. In addition to mapping the tasks on the same PE, our heuristics map the tasks of an application close to each other and thus try to reduce the distance between the source and destination PE, resulting in reduced average packet latency. Table 4.1 shows average packet latency for the first simulated scenario at varying communication rate (% usage of available bandwidth) when different heuristics are employed. It can be seen that PBN and PTBN provide almost the same average packet latency.

4.4.4 Effect of Time Bound

Heuristic PTBN has been analyzed for different performance metrics at varying allowed time bounds. The heuristic finds a better neighbor for each task as the allowed time bound increases. Selecting a better neighbor for mapping each task will help to distribute the loads in the channels (congestions) more homogeneously. Therefore, average channel load and packet latency decrease with the time bound. The allowed time bound gets added to the total execution time and thus nullifying the advantage of reduced congestion providing faster communication. The total execution time behavior at different time bound values has been analyzed in order to find trade-offs between the execution time and time bound.

For finding a mapping for each task, the provided time bound to PTBN varies from zero to the difference in mapping time by PBN and PNN. At lower values of time bound, PTBN behaves like PNN and like PBN at higher values of time bound. Providing a time bound of more than the difference value does not affect the performance as the algorithm gets terminated after evaluating all the neighbors.

Figure 4.5 shows total execution time for first simulated scenario when heuristic PTBN is employed at different values of time bound for three different considered platforms where each PE supports 2, 4 and 8 tasks. The results shown are for an injection rate of 15% (% usage of available bandwidth). Similar behavior is obtained at other injection rates for all the scenarios. A couple of observations can be made from Figure 4.5. First, total execution time decreases as platforms supporting larger number of tasks on each PE are considered. This is because of reduction in communication overhead by allowing mapping of larger number of communicating tasks on the same PE. Second, execution time for all the platforms first falls with the time bound and then starts increasing after reaching a minimum and finally becomes constant. The falling trend shows that reduction in communication time due to better neighbor selection is more than the allowed time bound. The increasing trend shows that the allowed time bound adds more to the total execution time than the reduction in the communication time. The constant region is obtained due to termination of the heuristic even before the allowed time bound just after evaluating all the neighbors. It can also be observed that as the platforms supporting larger number of tasks on each PE are considered, the falling trend, minimum and constant region are obtained earlier. This shows that the dependency on the best neighbor decreases with the platforms supporting larger number of tasks on each PE. It has also been seen that the difference in the maximum and minimum execution time decreases as platforms supporting larger number of tasks per PE are considered.

4.5 Summary

In this chapter, we have proposed mapping techniques for the run-time mapping of applications onto NoC-based heterogeneous MPSoC platforms containing multi-task sup-

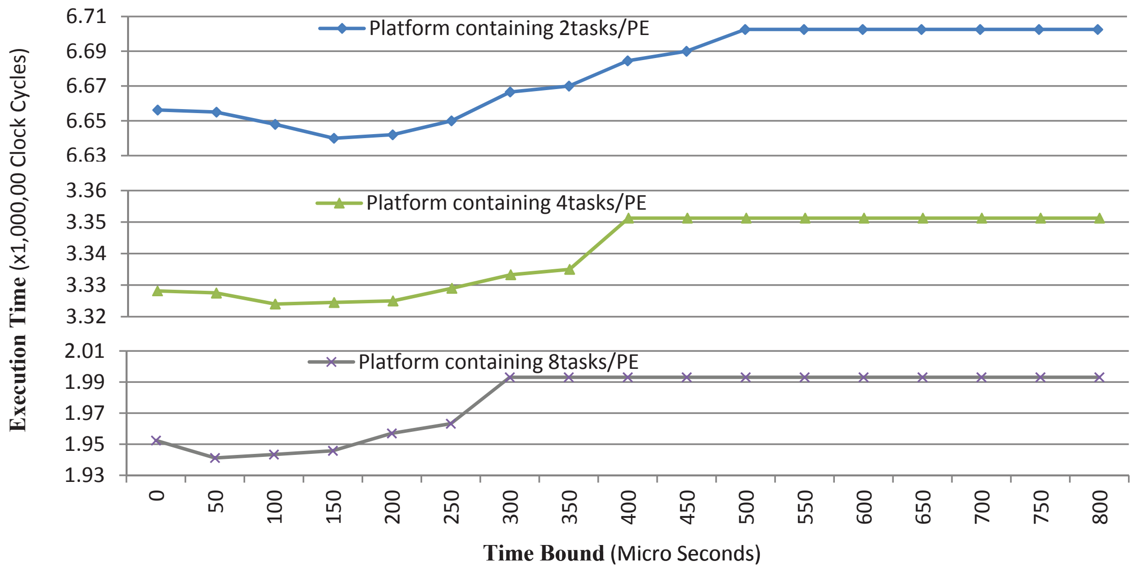


Figure 4.5: Execution time at varying time bound values

ported PEs. The techniques take advantage of multi-task supported PEs by mapping communicating tasks on the same PE whenever possible in order to reduce the communication overhead. This is possible as communicating tasks that are mapped on the same PE do not require any network resource for their communication, hence resulting in significant reduction in communication time. For each application, the first technique maps the tasks on the same PE or in close proximity and the second technique considers traffic in addition to the proximity of tasks. The traffic is reduced by considering the best neighbor from the available neighbors to map the tasks. However, the overhead in finding the best neighbor has significant impact on the total execution time. The third technique reduces the overhead by providing a time bound to evaluate the neighbors so that all the performance metrics are optimized simultaneously. When compared to MPSoC with single task supported PEs, the technique requires a lower evaluation time bound for obtaining optimal execution time and average channel load. This is possible as the evaluation needs to be performed only for tasks that are not mapped with their communicating task on the same PE. In addition, the difference in total execution time by the time bounded and non-time bounded approaches have decreased notably. Comparisons with existing techniques show that total execution time, average channel load and average packet latency have been reduced. In addition, we have shown that state-of-the-art run-time mapping techniques extended for multi-task supported PEs also achieve performance improvement.

Mapping techniques reported in the literature and those proposed here do not consider communication between tasks during mapping. They can map many of the non-communicating requested tasks on the same PE if they cannot be supported on their master task PEs due to resource availability issues. The requested non-communicating tasks can be mapped on separate PEs by considering communication between the tasks so that remaining requested tasks can be mapped with their master (communicating) tasks. Thus, communication overhead can be further reduced. In the next chapter, we present communication-aware mapping techniques that consider communication between tasks during mapping in order to further improve the performance metrics governed by the communication overhead.

Chapter 5

Communication-aware Mapping

In this chapter, communication-aware mapping techniques for the multi-task supported PEs are proposed and evaluated. Mapping techniques reported in the literature and those proposed in the previous chapters do not consider communication between tasks during mapping at run-time and thus are not capable of exploiting the PEs efficiently. Instead, they map many non-communicating tasks on the same PE and in turn the communicating tasks get mapped on separate PEs due to resource availability issues on the PEs. Several design-time mapping techniques that perform mapping in communication-aware manner are reported in literature [103] [155], but they can not handle dynamism incurred at run-time such as addition of a new application in the system.

The proposed communication-aware run-time mapping techniques for efficient mapping of applications onto MPSoCs have been shown to overcome the drawbacks of existing techniques. The proposed techniques examine the available resources prior to recommending the adjacent communicating tasks on to the same PE. In addition, they give priority to the tasks of an application to be mapped in close proximity so as to further minimize the communication overhead. Our investigations show that the proposed techniques consistently lead to reduction in the average channel load, total execution time, average packet latency and energy consumption. In particular, we show that energy savings can be up to 46% and average channel load is improved by 10% for some cases. The work discussed in this chapter has been published in [C-4] and [J-1].

The rest of the chapter is organized as follows. In Section 5.1, we provide an overview of the proposed communication-aware strategy and discuss their advantages over the existing approaches when applied to MPSoCs containing multi-task supported PEs. Efficient run-time mapping techniques based on the communication-aware strategy are presented in Section 5.2. In Section 5.3, performance of the techniques is evaluated and compared with existing techniques. We summarize the chapter in Section 5.4.

5.1 Communication-aware Strategy

In this section, we introduce our communication-aware strategy that take maximum advantage of the multi-task supported PEs by carefully mapping the communicating tasks on the same PE as far as possible. This result in reduced communication overhead and the tasks on the same PE can communicate faster as they do not need any network resource. The definitions of *application task graph*, *MPSoC architecture* and *mapping* are the same as described in Chapter 3.

Existing strategies do not map the communicating tasks in a highly communication-aware manner and thus are not able to efficiently utilize the multi-task supported PEs of MPSoCs. Figure 5.1 shows one possible mapping of an application on part of the MPSoC architecture by applying Nearest Neighbor (NN) run-time mapping strategy proposed in [129]. For the mapping example shown in Figure 5.1, each PE is assumed to support a maximum of three tasks. However, in practice, the PEs can support larger number of tasks and the results for varying number of tasks on each PE are presented in Section 5.3. For mapping the application task graph, first, the initial task (0) is mapped and other tasks are mapped at run-time when a communication to them is required. When the initial task starts its execution, it requests its communicating slave tasks (1 & 2) and their mapping is found on the nearest possible neighbor PE. As each PE is assumed to

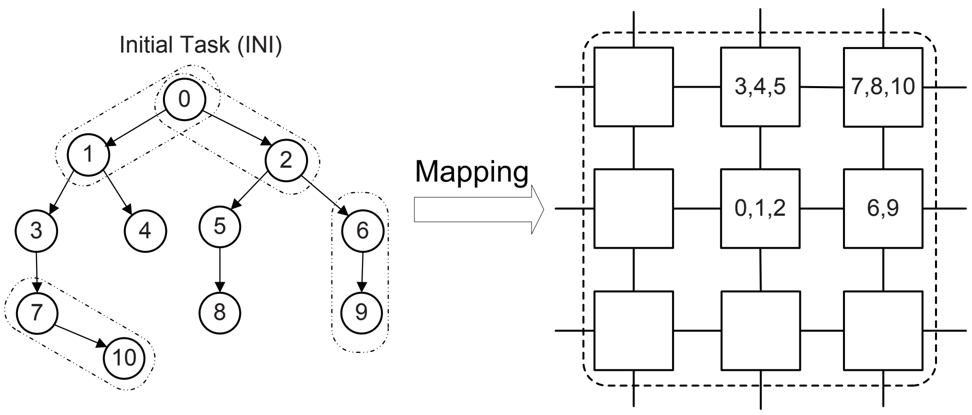


Figure 5.1: Mapping of an application by state-of-the-art mapping heuristic

support three tasks, the requested tasks (1 & 2) can be mapped onto the same position as task 0 (hop_distance = 0). After mapping tasks 1 and 2, they start their execution and their slave tasks (tasks 3 & 4 for task 1; tasks 5 & 6 for task 2) are requested and their mapping is found by the NN mapping strategy. The possible mapping for tasks 3, 4, 5 and 6 is shown. Now, when these tasks (3, 4, 5 & 6) start executing, their slave tasks (7, 8 & 9) are requested and their mapping is found. In the same manner, task 10 gets requested and mapped when task 7 starts its execution. A possible mapping for all the requested tasks of the application by NN strategy is shown in Figure 5.1. The communication between the communicating tasks start when they are mapped.

Communicating task pairs (0, 1), (0, 2), (6, 9) and (7, 10) as highlighted in Figure 5.1 get mapped onto the same PE. Thus, communication overhead gets reduced. However, the remaining communicating pairs need to communicate from different PEs, thus a lot of communication overhead still remains. Other existing approaches show similar behavior. The overhead can be further reduced if more communicating pairs can be mapped onto the same PE. Next, we provide an overview of our proposed communication-aware strategy that make it possible to reduce the communication overhead by a large amount.

In our proposed communication-aware strategy, the requested tasks are mapped by looking at the previously mapped tasks on the PEs. The placement for a requested task is searched in increasing hop distances (hop_distance = 0 to max_hop_distances) that results in mapping of all the tasks of an application close to each other. After finding the placement (PE) for the requested task, previously mapped tasks at the PE are found. If found PE does not have any previously mapped task then the PE is evaluated for mapping. Otherwise, the previously mapped task(s) are checked to have communication with the requested task. The requested task is mapped at the same position (PE) if they communicate; otherwise it is mapped onto the next possible position even if it is supported by the current PE. The same process is adapted for each requested task. This strategy forces mapping of the communicating tasks onto the same PE if they can be supported and avoids mapping of the non-communicating tasks onto the same PE. This process may cause some leaf tasks to occupy the whole PE without sharing it with other tasks if they are not mapped on the same PE as their masters. The leaf tasks don't have any slave that can be requested. However, in dynamic scenarios it may not be able to predict future tasks, so some leaf tasks can occupy the whole PE.

One possible mapping for the same application onto the part of the MPSoC is depicted in Figure 5.2. After mapping initial task (task 0), communicating tasks 1 and 2 are requested. Tasks 1 and 2 communicate with task 0 and are thus mapped on the same PE allowing for maximum three tasks. Now, tasks 1 and 2 request their communicating tasks 3&4 and 5&6 respectively. Task 3 is mapped on a neighboring PE to its master (task 1) as it cannot be mapped on the same PE that is fully occupied. Task 4 is also mapped onto a neighboring PE to its master (task 1). Unlike the NN strategy that maps task 4 with task 3, here, task 4 is mapped on a new PE as it does not communicate with task 3. The new PEs are chosen for mapping so that other communicating tasks can be mapped with their master tasks. Similarly, tasks 5 and 6 are mapped on a new

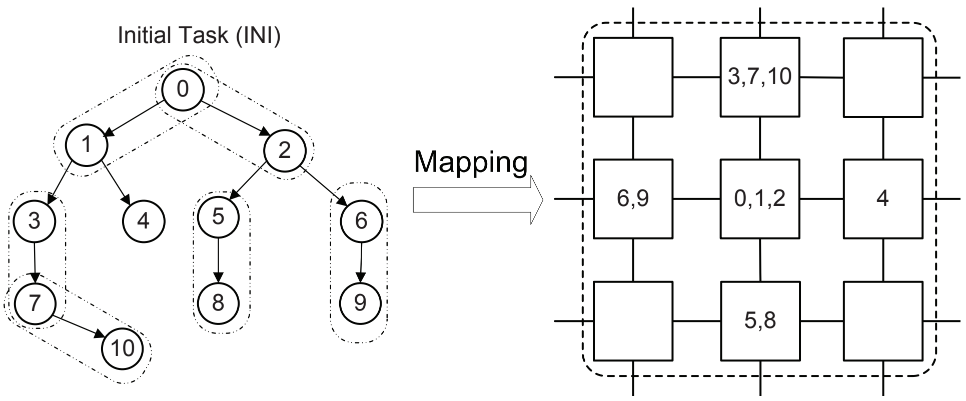


Figure 5.2: Mapping of application by communication-aware strategy

PE closed to their master tasks as shown in Figure 5.2. Now, when tasks 3, 4, 5 and 6 start executing, their slave tasks (7, 8 & 9) are requested to be mapped on the PEs. The tasks are tried to be mapped with their master (communicating) tasks in order to reduce the communication overhead. Tasks 7, 8 and 9 are mapped with their master tasks 3, 5, and 6 respectively. In the similar manner, task 10 gets requested when task 7 starts its execution. Task 10 is mapped with its master (communicating) task 7.

This strategy forces the mapping of most of the communicating pairs onto the same PE. The communicating task pairs (0, 1), (0, 2), (3, 7), (5, 8), (6, 9) and (7, 10) as highlighted in Figure 5.2 are mapped onto the same PE, resulting in reduced communication overhead. The leaf task 4 occupying the whole PE can be mapped on the PEs already containing some tasks in order to increase the resource utilization, but the communication overhead might get increased if the task gets mapped far from its master (communicating) task. So, we leave mapping of such leaf tasks as it is. The communication overhead is greatly reduced by mapping maximum communicating pairs onto the same PE.

A static mapping approach can provide better solution than dynamic mapping approaches provided the applications' structure and workload of their tasks is known at design-time. The static approach is performed at design-time with well known computation and communication behavior of tasks and resources status, enabling to explore better mapping decisions. However, the dynamic approaches are adequate for the scenarios where the applications' structure, their workload and resources status is available only at run-time. So, if applications are known at design-time, the static approach provides a better solution and thus an ideal mapping solution.

By knowing all the tasks at design-time, most of the communicating pairs can be mapped onto the same PE to reduce the communication overhead. In order to find an ideal mapping (containing maximum communicating pairs on the same PE), the whole application graph is partitioned into groups of connected tasks to 1) minimize the number of groups, and 2) maximize the number of tasks in each group that is limited by memory space available on the PE. The tasks of each group are then mapped on separate PEs. The ideal mapping provides optimal mapping in terms of number of communicating pairs to be mapped onto the same PE. The ideal mapping of an application onto the part of MPSoC is depicted in Figure 5.3. The communicating task pairs (0, 1), (1, 4), (2, 5), (5, 8), (3, 7), (7, 10) and (6,9) are mapped onto the same PE, providing a better solution than the dynamic mapping strategies where maximum number of communicating pairs are tried to be mapped on the same PE at run-time. Although, the ideal static mapping decisions provide better solution; these cannot be applied to dynamic scenarios where workload of tasks is unknown at design-time.

5.2 Run-time Mapping Heuristics

This section details mapping heuristics developed with the communication-aware mapping strategy introduced in the previous section.

Applications' mapping is started by first mapping the initial tasks in a distributed manner by dividing the NoC into clusters in the similar manner as described in Figure

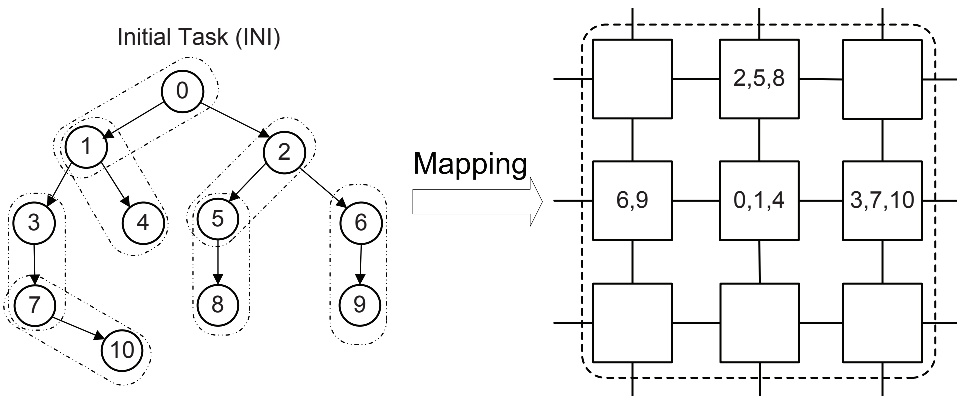


Figure 5.3: Ideal mapping of application with static mapping decision

4.1 of Chapter 4. When the initial tasks start their execution, communicating tasks are requested and their mapping is found.

To find the placement for a requested task, first it is checked to see if there is any free resource in the platform, having the same type (hardware, software or initial) as of the requested task. If yes, then placement would be found by scanning the whole network in increasing hop distances starting from the requesting node (hop_distance = 0) to the max_hop_distance (NoC limit). The NoC limit is the maximum hop distance up to which the network can be scanned to find the mapping. The maximum hop distance is the *Manhattan* distance from starting point of the scan to the farthest boundary corner of the NoC. If there is no supported (same type) free resource in the platform then the task is entered into its corresponding queue and waits until a supported resource becomes free. The queued task is mapped on the freed supported resource and the control is transferred to find the placement for another unmapped requested task after updating the resources' status. The same strategy is applied to all the tasks whenever they get requested. The requested tasks can be mapped by the following developed mapping heuristics.

5.2.1 Communication-aware Packing-based Nearest Neighbor

This algorithm is explained through Algorithm 4. At each hop distance, PEs list is found by function ***get_packing_ordered_list***(hop_distance) (step 5), where PEs are selected in left, down, top and right order as in the PNN heuristic proposed in the previous chapter. Now, the PEs are evaluated in their selection order, to find the most suitable PE for the requested task. If the selected PE can support the task (step 7) then previously mapped tasks onto the PE are found (step 8). If there is no previous mapped task (i.e. first task onto the PE) (step 13) then the requested task is mapped onto the PE (step 14); otherwise previously mapped tasks are checked to have communication with the requested task (step 10). If they are communicating, then only the task is mapped onto the PE (step 11); otherwise next possibility is evaluated so that the PE can accommodate another task having communication with the already (previously) mapped one. After finding the mapping, the selection and evaluation process is stopped (step 11 and 14) even if there might be another supported PE at the same hop distance. Resources are updated after mapping in order to have accurate information about their occupancy for other requested tasks. This heuristic maps tasks of each application almost similarly as in Figure 5.2.

By choosing left and down side PEs first, the PEs of top and right edge of the cluster (Figure 4.1 of Chapter 4) are intentionally made free so that these can be used by tasks of other applications running on top and right side. This way all the applications' tasks are mapped in bottom-left fashion utilizing the PEs on the top and right edge of the cluster from other applications. Thus, resource utilization increases that results in improved performance.

5.2.2 Communication-aware Packing-based Best Neighbor

This algorithm is a combination of path load computation approach and the CPNN algorithm and is presented through Algorithm 5. Path load computation approach is

Algorithm 4: Communication-aware Packing-based Nearest Neighbor (CPNN)

```
Input: TG(T,E), AG(P,V) // task $t_i \in T$ ; PE $p_i \in P$ (PE)
Output: mpg (mapping $TG(T,E) \rightarrow AG(P,V)$ )
   // NFR[type]: number of free resource(s) of type type in NoC
 1: Map the initial task ($INI \in T$) at the centre of the cluster;
 2: for all unmapped task $t_i \in T$ that is requested do
 3:   if $NFR[ti_{type}]$ != 0 then
 4:     for all hop_distance = 0 to max_hop_distance do
 5:       PE_list = get_packing_ordered_list(hop_distance);
 6:       for all PEs $\in$ PE_list do
 7:         if $ti_{type}$==$pi_{type}$ AND resource available at $p_i$ then
 8:           Find previously mapped tasks on $p_i$;
 9:           if previous_tasks != NULL then
10:             if comm[$t_i$][any_previous_task] then
11:               Map $t_i$ onto PE $p_i$ and exit to step 24;
12:             end if
13:           else
14:             Map $t_i$ onto PE $p_i$ and exit to step 24;
15:           end if
16:         end if
17:       end for
18:     end for
19:   else
20:     insert($t_i$ to Queue($ti_{type}$));
21:     wait until $NFR[ti_{type}]$ != 0;// updated at run-time
22:     release($t_i$ from Queue($ti_{type}$));
23:     Map $t_i$ onto the freed resource at node $p_i$;
24:     insert($p_i$ to mpg); update(resources by mpg);
25:     wait and goto step 3 if new task $t_i \in T$ is requested;
26:   end if
27: end for
```

incorporated by calculating the path load (step 15) for each PE (from Equation Eq. 3.1 of Chapter 3). In CPNN algorithm, at each hop distance, if any evaluated PE is suitable for the requested task then it is selected for mapping and other PEs at the same hop distance are not evaluated. In contrast to CPNN, here, all the PEs are evaluated (selected temporarily- step 18) and finally, the PE with minimum path load is considered for final mapping. The same strategy is followed at each hop distance until placement for the requested task is found.

Algorithm 5: Communication-aware Packing-based Best Neighbor (CPBN)

```
Input: $TG(T,E)$, $AG(P,V)$ // task $t_i \in T$ ; PE $p_i \in P$ (PE)
Output: $mpg$ (mapping $TG(T,E) \rightarrow AG(P,V)$ )
 1: Map the initial task ($INI \in T$) at the centre of the cluster;
 2: for all unmapped task $t_i \in T$ that is requested do
 3:   if $NFR[ti_{type}]$ != 0 then
 4:     for all hop_distance = 0 to max_hop_distance do
 5:       weight = MAX_VALUE; // some large value
 6:       PE_list = get_packing_ordered_list(hop_distance);
 7:       for all PEs $\in$ PE_list do
 8:         if $ti_{type}$==$pi_{type}$ AND resource available at $p_i$ then
 9:           Find previously mapped tasks on $p_i$;
10:           if previous_tasks != NULL then
11:             if comm[$t_i$][any_previous_task] then
12:               Map $t_i$ onto PE $p_i$ and exit to step 29;
13:             end if
14:           else
15:             weightTemp = calcChannelLoad(when $t_i$ mapped onto $p_i$);
16:             if weightTemp < weight then
17:               weight = weightTemp;
18:               Select node $p_i$ temporarily to map $t_i$;
19:             end if
20:           end if
21:         end if
22:       end for
23:       if weight < MAX_VALUE then
24:         Map $t_i$ onto PE $p_i$ and exit to step 29;
25:       end if
26:     end for
27:   else
28:     Perform steps 20 to 23 from Algorithm CPNN;
29:     insert($p_i$ to $mpg$); update(resources by $mpg$);
30:     wait and goto step 3 if new task $t_i \in T$ is requested;
31:   end if
32: end for
```

This algorithm considers traffic (congestion in channels) while finding the placement for a requested task, hence it is a congestion-aware mapping heuristic that tries to distribute the channel load more homogeneously in the NoC. Additionally, it takes advantage of the packing strategy and communication-aware strategy.

5.3 Performance Evaluation

In this section, we evaluate performance provided by the proposed mapping techniques. The evaluated performance metrics are *total execution time*, *energy consumption*, *average channel load* and *average packet latency* for varying set of applications.

The simulation platform used to perform the experiments is the same as in Chapter 4, where each processing node supports more than one task. We have performed the simulation by varying the number of tasks to be supported at each node. However, it is known that larger memory space and reconfigurable area will be required to support more number of software and hardware tasks respectively. So, we have performed simulation up to a maximum four tasks per node in order to avoid the memory space and reconfigurable area availability issues.

Application modeling is done similar to the previous chapters, with an initial task, hardware tasks and software tasks. The packets transmission between the tasks takes place in the same manner. The processing time of packets corresponding to each task mapped on the same PE is proportional to the number of tasks mapped on the PE as the packets for each task are served in time multiplexed manner. The simulation is performed at varying processing time to analyze the computation-communication behavior.

The evaluated scenarios are:

(i) *20 identical Tree like Applications having all tasks as software tasks*: (parallel benchmarks have this profile), each having 10 tasks, where one task is taken as initial (starting task) and rest 9 as software tasks.

(ii) *20 identical Tree like Applications having hardware and software tasks*: each having 10 tasks, where one task is taken as initial, 2 as hardware and rest 7 as software tasks.

(iii) *20 random applications having hardware and software tasks*: random generated applications using *Task Graph For Free* (TGFF[92]). Each has one initial and random number of hardware/software tasks (varying from 4 to 9).

In the first two scenarios, simulation is performed with injection rate varying from 5 to 20% (% usage of available channel bandwidth) and in third it is random from 5 to 30%.

For scenario (i) evaluation, 8×8 NoC-based homogeneous MPSoC is taken, where all the PEs are processors. For evaluating scenarios (ii) and (iii), 8×8 NoC-based heterogeneous MPSoC is taken with 52 nodes as processors and 12 nodes as reconfigurable areas. In all the scenarios, one software node is used for the Manager Processor (M) that is considered to support a single task. Scenarios (i), (ii) and (iii) have also been referred to as scenarios 1, 2 and 3 respectively.

The number of simultaneously running applications (initial tasks) is increased as processing capability of the platform is increased by supporting more number of tasks at each platform PE. This is required for maximum utilization of all the platform resources. For the considered MPSoC platforms containing 2, 3 or 4 tasks supported PEs, the best results are obtained at 10, 14 and 18 simultaneously running applications respectively.

The initial task placement is done by a clustering approach, where the processing capability of a cluster is determined by the non-shared PEs within the cluster. The processing capabilities of clusters are stored in advance. The applications to be mapped are sorted in descending order by the number of tasks in them, before the actual mapping starts. The initial tasks of the sorted applications are mapped on the center of the clusters sorted in their decreasing processing capability. Thus, an application containing more number of tasks is tried to be mapped into a more processing capability cluster and an application containing relatively less number of tasks into a lower processing capability cluster, resulting in better resource utilization. This approach is not useful for the scenarios where all the applications contain same number of tasks like scenario (i) and (ii). Scenario (iii) contains applications with varying number of tasks in them so the approach is useful.

The mapping heuristics NN & BN proposed in [129], PNN & PBN proposed in the previous chapter, CPNN, CPBN and the ideal static mapping (ISM) decision are evaluated on three different platforms that contain PEs supporting two, three or four tasks.

5.3.1 Total Execution Time

It is the time taken to finish the execution of all the applications, which is computed in similar manner as in the previous chapters. The total execution time mainly consists of computation and communication time of the packets. However, if the computation time is less than the time interval between receiving two consecutive packets corresponding to the same task on the PE then the computation time will get absorbed within the communication time. Thus, the computation time should be large enough in order to contribute to the total execution time.

The proposed mapping heuristics map the maximum number of communicating pairs onto the same PE, resulting in reduced communication overhead and the traffic in channels (channel congestion) that gets generated when communicating pairs communicate from different PEs. The reduced traffic decreases the communication overhead of other communicating tasks that communicate from different PEs. The communicating pairs mapped on the same PE can communicate faster and the reduced traffic facilitates for the faster communication of the communicating tasks mapped on different PEs. Therefore, communication time is reduced and thus the total execution time.

Figure 5.4 shows average execution time required for the first simulated scenario at different platforms when heuristics NN, PNN, CPNN and ISM are employed. The average for each heuristic is taken after executing it at varying injection rate. A couple

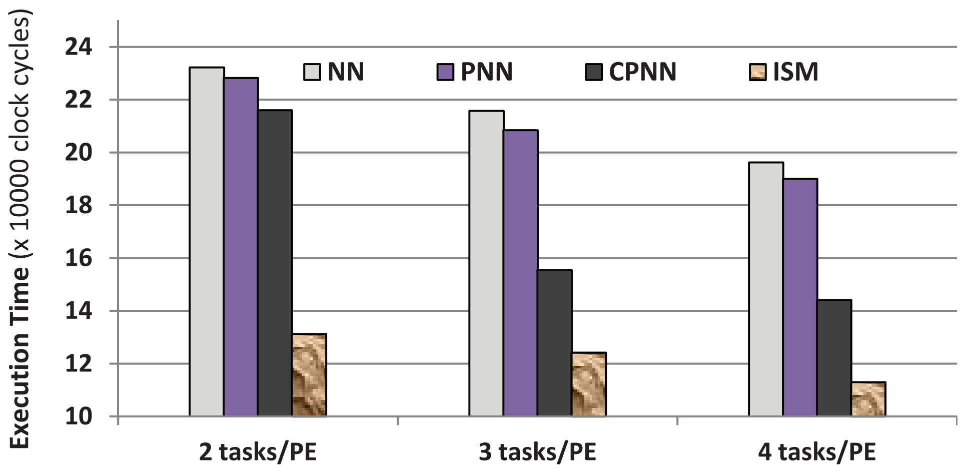


Figure 5.4: Execution time for NN, PNN, CPNN and ISM heuristics at different platforms

of observations can be made from the Figure 5.4. First, PNN always perform better than NN. Second, CPNN performs better than NN at each platform and thus are scalable for platforms containing PEs to support even higher number of tasks. Third, the largest gain for CPNN over NN is witnessed for 3 tasks/PE platform. For this platform, CPNN shows an average gain of 27.93%. Forth, ISM outperforms all the heuristics at each platform as the mapping decision is taken at design-time with a global view of the platform resources and takes maximum advantage from the task graph structure. However, this cannot be applied to dynamic scenarios. It has also been observed that gain of CPBN over NN is 27.03%. CPBN shows lesser gain as it searches for the best neighbor for each task and the searching time gets added to the total execution time.

The proposed heuristics have been analyzed for their time complexity. All the heuristics have time complexity of the same level and is of the order of *O(C)*, where C is the number of PEs in the platform. Thus, the heuristics execute almost in similar time.

5.3.2 Energy Consumption

Energy is required when a packet needs to be transmitted from source PE to destination PE and then to process the packet at the destination PE after it is received. The energy required in transmission and processing are referred to as communication and computation energy respectively.

The communication energy depends on the number of bits to be transmitted, the number of links to be traversed between both the PEs and energy required in transmitting each bit through one link. The transmitted bits are calculated by multiplying number of packets by the average packet size in bits. Here, number of packets is considered as data volume V_{ms} and average packet size as ten flits each of 16 bits denoted as P_{size}, when transferred from master to slave. As communication takes place from slave to master too, so the total bits include the number of packets transferred from slave to master as well and the number is considered as V_{sm} having the same average packet size. The number of links to be traversed between the source and destination PE is calculated from the Manhattan distance $(\Delta X_{ms} + \Delta Y_{ms})$ between the PEs as XY routing algorithm is used. The energy required to transmit one bit through each link is considered as E_{Lbit} [156] [157]. The E_{Lbit} value used in this work has been estimated as the product of power required to transfer one bit through each link (11.26mW) and transfer delay. The communication energy is estimated as product of number of bits to be transmitted, the number of links to be traversed between source and destination PE and the energy required to transmit one bit through one link, for each master-slave pairs from Equation Eq. 5.1.

$$E_{comm} = \sum[(V_{ms} + V_{sm}) \times P_{size} \times (\Delta X_{ms} + \Delta Y_{ms}) \times E_{Lbit}] \qquad \text{(Eq. 5.1)}$$

The computation energy depends on the number of bits to be processed on the receiver PE, time required to process each received bit and power needed to process the bit. The bits to be processed are the same that were transmitted from some source PE and are calculated by multiplying number of packets (V_{ms}) by the average packet size (P_{size}), when received by the slave. The total bits for each master-slave pair includes the bits to be processed on the master PE ($V_{sm} \times P_{size}$) as well, when received by master and sent by slave. The time required to process each bit is calculated by dividing the time taken to process each packet (t_{comp}) by the average packet size (P_{size}). The value of t_{comp} is provided by a configuration file. The power needed to process the bits on a PE is estimated from the power efficiency of Tile64 processor [158]. In [158], power efficiency is varied from 15-22W when all the 64 PEs operate at 700MHz simultaneously. The power is scaled for one PE operating at 25MHz and is referred as PE_{power}. The scaling is done for 25MHz as the NoC [64] also operates at 25MHz and it is very reasonable for the PEs to operate at the same frequency. An average power dissipation of 20W is considered while scaling is performed. The computation energy is estimated as product of the number of bits to be processed, time required to process one bit and power needed to process the bit, for each master-slave pairs from Equation Eq. 5.2.

$$E_{comp} = \sum [(V_{ms} + V_{sm}) \times t_{comp} \times PE_{power}] \qquad \text{(Eq. 5.2)}$$

Total energy consumption is estimated as the sum of communication and computation energy from Equation Eq. 5.3. Our proposed mapping strategy reduces the distance between source and destination PE by placing the communicating tasks onto the same PE, where bits can be exchanged very easily through some common memory or register without the need of much communication energy. Thus, total energy consumption is greatly reduced.

$$E_{total} = E_{comm} + E_{comp} \qquad \text{(Eq. 5.3)}$$

Figure 5.5 shows energy consumption (E_{total}) for all the simulated scenarios at different platforms when heuristics PBN and CPBN are employed. The energy for heuristics PNN and CPNN is not shown as it will be lesser than PBN and CPBN respectively.

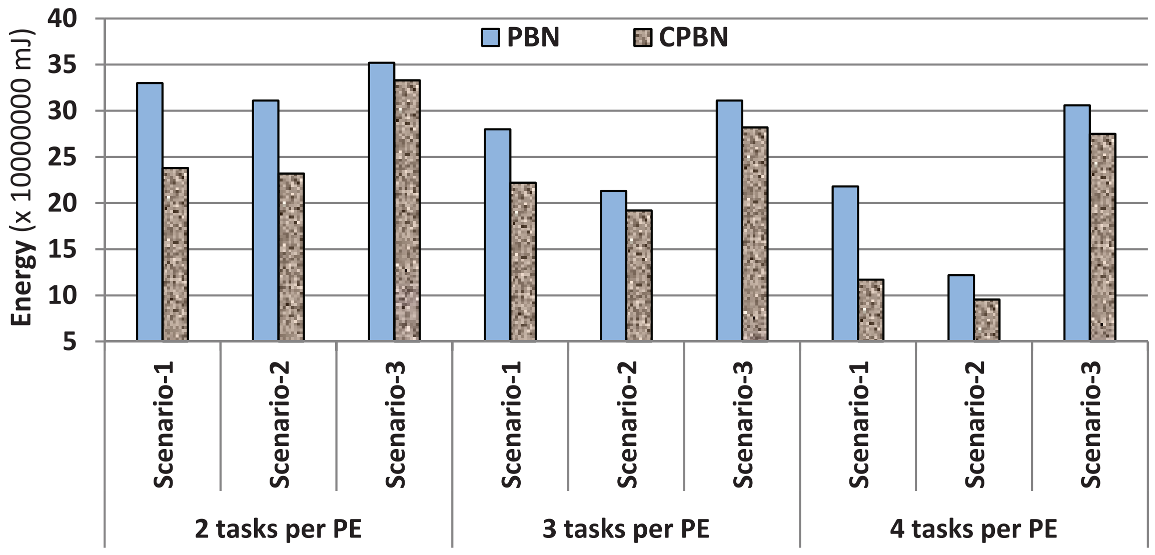


Figure 5.5: Energy consumption for PBN and CPBN heuristics in different platforms for all scenarios

The reason behind this is that PBN and CPBN try to distribute traffic in the channels uniformly and thus facilitate for lower energy consumption during communication. A number of observations can be made from the Figure 5.5. First, CPBN always performs better than PBN and maximum gain of CPBN over PBN is witnessed for Scenario-1 at each platform. At 4 tasks per PE platform, CPBN shows an improvement of 46.3% over PBN. Second, the energy consumption for all the heuristics is lowest for Scenario-2 at each platform when compared with other scenarios. Therefore, CPBN needs to be employed for optimizing energy consumption.

5.3.3 Average Channel Load

The average channel load represents the NoC use and is calculated in the similar manner as in previous chapters. The load in the channels that depends upon the traffic produced by the communicating tasks is reduced by the proposed mapping heuristics by mapping maximum number of communicating pairs onto the same processing node. Thus, average

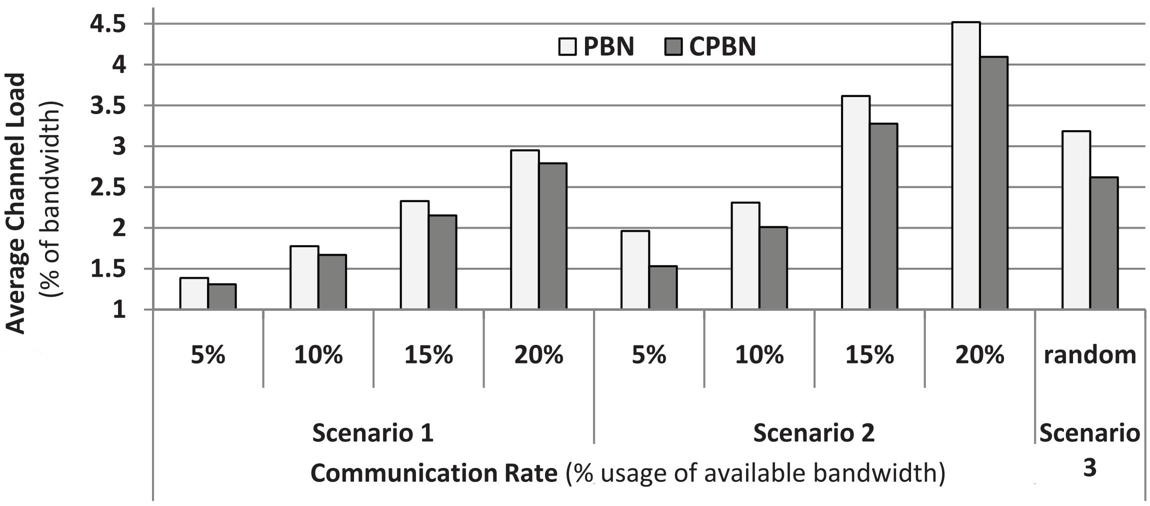


Figure 5.6: Average channel load for PBN and CPBN heuristics for all simulation scenarios

channel load is significantly reduced when proposed heuristics are employed.

Figure 5.6 plots average channel load for all simulation scenarios at varying injection rates, for 3 tasks per PE platform. When executing for other platforms, a similar behavior is obtained. The average channel load increases with communication rate as more traffic gets generated in the channels with increase in the communication rate. CPBN reduces the average channel load for all the scenarios when compared to other heuristics. CPBN shows an average gain of 10.67% over PBN.

5.3.4 Average Packet Latency

The average packet latency is calculated in the similar manner as described in the previous chapter. It depends on the congestion in the path and the distance between the source and destination PE on which communicating tasks are mapped. The proposed mapping heuristics try to map the maximum communicating task pairs onto the same processing node, reducing the traffic produced (congestion) in the channels. The reduced congestion

Scenarios		**Average Packet Latency** (Clock Cycles)			
	Rate	BN	PBN	CPBN	ISM
Scenario 1	**5%**	118	115	113	110
	10%	216	216	216	211
	15%	316	314	309	303
	20%	419	414	405	411
Scenario 2	**5%**	116	116	116	105
	10%	214	214	213	212
	15%	310	309	307	303
	20%	410	407	403	397
Scenario 3		288	287	275	NA
% Gain		–	**0.62%**	**2.08%**	

Table 5.1: Average Packet Latency for all simulated scenarios when BN, PBN, CPBN and Ideal Static Mapping (ISM) mapping decision are employed.

helps in reducing packet latency for other tasks communicating from different PEs. Thus, proposed mapping heuristics reduce the average packet latency.

Table 5.1 shows the latency results for all simulated scenarios for two tasks per PE platform. Communication-aware mapping heuristic CPBN reduce the average packet latency when compared to BN and PBN. Improvements are shown as *% Gain* in the last row of table 5.1 for PBN and CPBN over BN. Improvements are not significant as packets for the communicating tasks mapped on the same PE are not considered. ISM performs better than all other heuristics for Scenario-1 and Scenario-2 but cannot be applied to Scenario-3 as applications' structure and their workload are random and not known at design-time. Other evaluated platforms show almost similar results.

5.3.5 Effect of Computation-Communication Ratio

The Computation-Communication Ratio (CCR) is estimated as the ratio of desired computation time (in cycles) and desired communication time (in cycles) for all the packets from the following equation, where $t_{computation}$ and $t_{communication}$ are the desired computation and communication time for individual packets.

$$CCR = \sum[t_{computation}] \div \sum[t_{communication}] \qquad \text{(Eq. 5.4)}$$

The number of packets to be transferred (communicated) and processed (computed) remains the same as all the transferred packets need to be processed at some PE. Since every packet is considered identical, each has the same desired computation time ($t_{computation}$) that is provided through a configuration file, and the same desired communication time ($t_{communication}$) that is calculated for NoC [64] operating at 25 MHz for a given injection rate (% usage of available bandwidth). Thus, CCR simply reduces to the ratio of $t_{computation}$ and $t_{communication}$ due to the same number of identical packets. The value of $t_{communication}$ remains fixed for a given rate. Therefore, in order to get varying values of CCR, different values of $t_{computation}$ (clock cycles) are provided through the configuration file.

The total execution time mainly consists of computation and communication time. The computation of a packet starts just after it is received on a PE and gets finished before receiving the next packet. So, the computation time should be greater than the time interval between receiving of two consecutive packets in order to contribute to the total execution time.

Figure 5.7 shows the total execution time for mapping heuristics NN and CPNN at varying CCR when applied to the first scenario at an injection rate of 5% (% usage of available bandwidth). The execution time behavior is shown for 2 tasks per PE platform. Other platforms also show almost similar behavior. It is clear that CPNN always performs better than NN. The gain by CPNN over NN varies for different CCR. The gain behavior for different platforms is described subsequently.

Figure 5.8 shows gain (%) in total execution time for mapping algorithm CPNN over NN when applied to the first scenario for different platforms at varying CCR. The gains

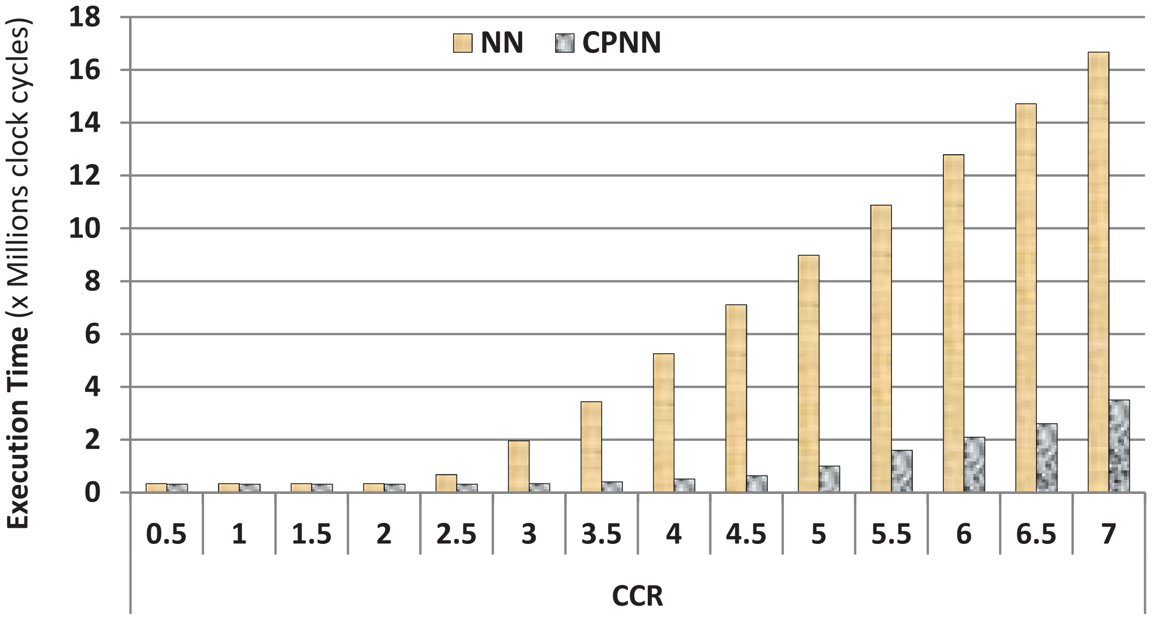


Figure 5.7: Total execution time for NN and CPNN at varying CCR for 2 tasks per PE platform

shown are for the injection rate of 5%. A couple of observations can be made from the Figure 5.8. For 2 tasks per PE platform, the gain is constant for some initial values of CCR. For these CCR values, the total execution time for both the algorithms (NN and CPNN) remain fixed as different values of computation time for each packet gets absorbed in the time intervals between receiving of consecutive packets. The constant gain is due to the communication time saved by employing the CPNN algorithm to map most of the communicating task pairs on the same PE. The communication time is saved as the packets for most of the communicating tasks are processed on the same PE without sending to any other PE. With further increment in CCR, we see continuous gains up to some CCR values and then a falling trend. The initial continuous gain is very drastic as increase in computation time adds to total execution time very much for NN when compared to the CPNN. The NN gets affected to a great extend as the computation times for most of the packets don't get absorbed between the time interval

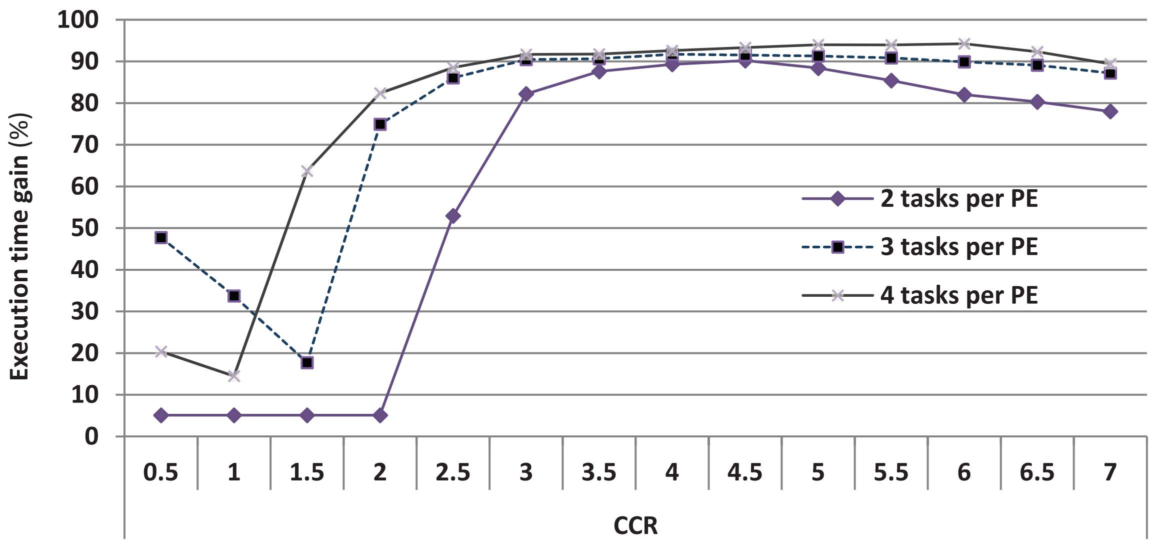


Figure 5.8: Improvement in total execution time for CPNN over NN at varying CCR for different platforms

of receiving consecutive packets, whereas in CPNN, most of the packets are exchanged on the same PE. Therefore, the total execution time for CPNN is not affected much as the computation time for the packets of the communicating tasks mapped on the same PE gets absorbed within the communication and computation time of other tasks' packets processed in parallel. The falling trend at higher values of CCR is obtained when the computation time starts adding to the total execution time significantly for the packets of the communicating tasks mapped on the same PE too for the CPNN heuristic. In the falling trend region, for CPNN, the computation time does not get absorbed within the computation and communication time of other parallely processed packets and thus the total execution time gets affected by all the packets for both the heuristics.

For 3 tasks per PE platform, the gain falls for some initial values of CCR and then shows a similar trend as in the case of 2 tasks per PE platform. For initial values of CCR, increase in computation time starts affecting total execution time for the CPNN heuristic as the computation time for the packets of communicating tasks mapped on the same PE does not get absorbed within the communication and computation time of other tasks' packets due to longer time required to process the packets. The time required in processing gets longer as the packet for a particular task is processed for some fixed clock cycles in time multiplexed manner along with packets of other tasks mapped on the same PE. Additionally, here, more number of tasks gets mapped on a PE. However, the total execution time for NN remains same for smaller values of CCR as computation time for these values of CCR does not add to the total execution time due to the same reason as in case of 2 tasks per PE platform. Thus, a falling trend is obtained for some initial values of CCR. The reason for the similar trend after some values of CCR is the same as explained for 2 tasks per PE platform. The drastic gain starts relatively at lower CCR values as compared to the 2 tasks per PE platform due to more tasks getting mapped on the PEs. For 4 tasks per PE platform, a similar trend is obtained with drastic gain starting relatively earlier due to the similar reason.

The initial gain for 3 tasks per PE platform is higher than other platforms. The almost flat region in mid values of CCR is longer for platforms supporting more number of tasks. Other heuristics also show similar behavior.

5.3.6 Clustering vs. Non-clustering

The clustering approach benefits the evaluated metrics only in the third simulation scenario, where applications contain varying number of tasks. In the clustering approach, first, applications are sorted by the number of tasks within them and then initial tasks of applications are mapped at the center of the clusters sorted by their processing capability as explained earlier. In non-clustering approach, applications are not sorted and their initial tasks are mapped at any random location.

Figure 5.9 shows gain obtained by the clustering approach over the non-clustering approach for average channel load, energy consumption, average packet latency and total

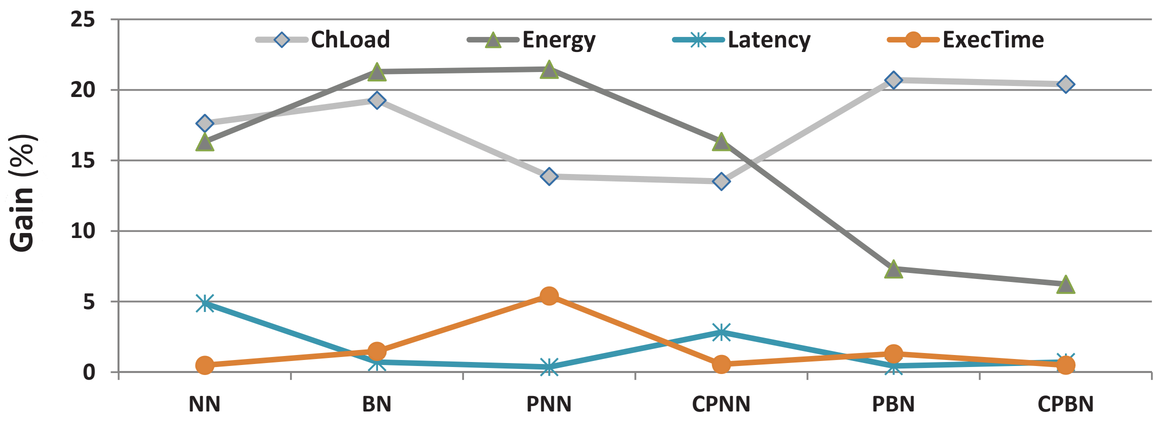


Figure 5.9: Improvements in clustering approach when different mapping algorithms are employed

execution time at the two tasks per PE platform when different mapping heuristics are employed. Average channel load and energy consumption is improved by around 15% with some improvement in packet latency and execution time. Similar behavior is obtained at other evaluated platforms too.

5.4 Summary

In this chapter, we have proposed run-time communication-aware mapping heuristics that try to map maximum communicating task pairs on the same PE in order to reduce communication overhead. The communicating tasks mapped on the same PE can communicate faster as they do not need any network resource. It has been observed that the communication-aware heuristics provide better results when compared to existing and earlier proposed heuristics. A simulation platform supporting more than one task on each PE has been considered. The PEs can be either a processor or a reconfigurable hardware (RH) block.

We show that the ideal static mapping solution can lead to performance improvement. The ideal static mapping has been shown to improve the overall performance when all the applications and their workloads are known prior to the mapping process. However, this does not cater for realistic scenarios in which the run-time characteristics call for dynamic mapping strategies.

Based on our investigations all the proposed heuristics have been evaluated using an 8×8 NoC-based MPSoC platform. We clearly demonstrate that the presented heuristics can consistently provide for notable reduction in the communication overhead. The potential of mapping adjacent communicating tasks and those of the same application onto the same PE whenever possible has contributed to the overall reduction in the communication overhead. We have investigated different scenarios depending on performance metrics of interest and show that the improvement in the total execution time can be up to 90% when the packet execution time is increased.

The communication-aware mapping strategies do not consider computation load balancing while reducing communication overhead. Therefore, they cannot perform well for the scenarios where computation overhead dominates communication overhead. In the next chapter, we present computation and communication aware mapping strategies that perform well in such scenarios.

Chapter 6

Computation and Communication Aware Mapping

Computation and communication aware mapping is desirable for scenarios where computation overhead dominates communication overhead or both the overheads become significant. In the previous chapter, we presented communication-aware mapping strategies that try to map maximum communicating task pairs on the same PE in order to reduce communication overhead. These techniques perform well when computation overhead is not of significant importance as compared to the communication overhead. However, when both the overheads are significant, mapping strategies taking care of both the overheads should be investigated in order to optimize the performance.

In this chapter, we propose strategies that efficiently map the application tasks on MPSoC processing elements (PEs) by taking computation overhead, communication overhead and resource utilization into account. The proposed strategies attempt to balance the computation load on the PEs while reducing communication overhead by mapping highly communicating tasks on the same PE. Based on our evaluations to map applications with varying number of tasks onto NoC-based MPSoCs, we show that the proposed strategies outperform the communication-aware strategies. In particular case study for mapping multiple MPEG-4 applications, we show that total execution time, energy consumption and resource usage are reduced by 33%, 39% and 37% respectively when compared to communication-aware mapping strategies. The work in this chapter has been published in [C-6], [C-8] and [C-10].

The rest of the chapter is organized as follows. Section 6.1 introduces some preliminaries necessary for proper understanding of the computation and communication aware mapping. In Section 6.2, we introduce our computation and communication aware run-time mapping strategies targeting homogeneous and heterogeneous MPSoC architectures respectively. In Section 6.3, performance of the techniques is evaluated and compared with communication-aware strategies. We summarize the chapter in Section 6.4.

6.1 Preliminaries

The definition of *application task graph* described in Chapter 3 is modified in order to model behavior of real-life streaming applications. Similar to Chapter 3, an application is represented as a directed task graph $TG = (T, E)$, where T is the set of application tasks and E is the set of all edges in the application. The edges connect the tasks and represent their communication. Here, each task is considered to be executable on both software and hardware resources, having different execution time on them. The difference in execution time on the resources determines suitability of a task to be executed on hardware or software resource. Figure 6.1 shows an example application task graph "A". A task $t_i \in T$ is represented as $(t_{id}, t_{hwcomp}, t_{swcomp})$, where t_{id} is the task identifier, t_{hwcomp} is the task hardware computation load in cycles when the task is executed on reconfigurable area and t_{swcomp} is the task software computation load in cycles when the task is executed on general purpose processor (GPP). An edge $e_i \in E$ represents the connection between two tasks and contains communication information between the tasks. The communication information is denoted as t_{comm}, which represents number of cycles taken for transferring the data volume for a single packet between the communicating tasks when full channel bandwidth is available. Similar to earlier chapter's definition, the communicating tasks

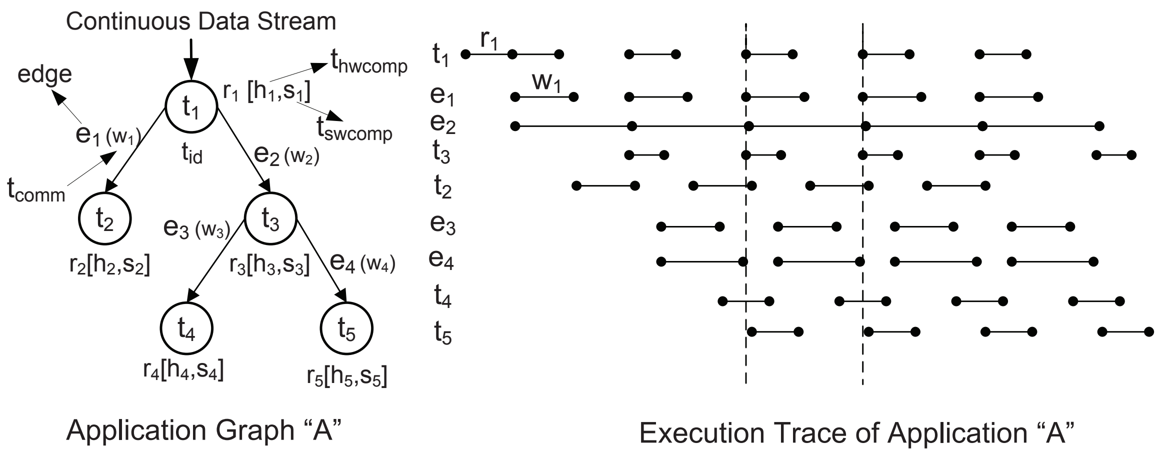


Figure 6.1: An example application and its execution trace

form a master-slave pair and the master task remains active till the last packet is sent to its slave tasks. A slave task starts executing once it has received a complete packet from its master task. For master-slave example, t_1 is master and t_2 & t_3 are slaves as shown in Figure 6.1.

The definitions of *MPSoC architecture* and *mapping* are the same as described in Chapter 3. Similar to previous chapter, each processing node of the MPSoC supports multiple tasks and channels support varying bandwidth (% usage of available bandwidth). However, for evaluation purpose, communicating tasks have been allocated full available channel bandwidth.

6.2 Computation and Communication Aware Mapping Strategies

In this section, we present computation and communication aware mapping strategies that consider computation load balancing while reducing communication overhead between tasks. First, we analyze communication-aware mapping strategies in order to find their drawbacks when applied to scenarios where both computation and communication overhead are significant. Then, we introduce our proposed strategies to overcome the drawbacks of communication-aware strategies.

State-of-the-art communication-aware mapping strategies do not consider computation load balancing while reducing communication overhead between tasks and thus are not able to perform well. One possible mapping of an application task graph on part of the MPSoC architecture is shown in Figure 6.2, which is obtained by applying the communication-aware mapping technique CPBN proposed in the previous chapter. This technique has been considered as the reference technique because it optimizes for performance metrics energy consumption, channel load and packet latency by the maximum amount with some improvement in total execution time too when compared to the existing techniques as discussed in the previous chapter. It has been assumed that each PE acts as a GPP and can support maximum four tasks. First, the initial task (task 0) is mapped. When task 0 starts executing, it requests it's communicating slave tasks (1 & 2). Tasks 1 and 2 communicate with task 0 and are thus mapped on the same PE allowing for a maximum of four tasks. Now, tasks 1 and 2 request their communicating tasks 3&4 and 5 respectively. Task 3 is mapped with its master task (task 1) as the PEs can support four tasks. Task 4 and 5 are mapped on neighboring PEs to their masters (task 1 for task4; task 2 for task 5) according to the CPBN strategy. The strategy maps them on two separate PEs by evaluating the neighboring PEs as shown in Figure 6.2. Now, when tasks 3, 4 and 5 start executing, their slave tasks (6, 7, 8, 9 & 10) are requested to be mapped on the PEs. The tasks are tried to be mapped with their master (communicating) tasks in order to reduce the communication overhead. Tasks 7, 8, 9 and 10 are mapped with their master tasks and task 6 is mapped on a separate PE close to its master. In the similar manner, tasks 11 and 12 get requested when task 9 starts its execution. Task 11 is mapped with its master (communicating) task 9. Task 12 is mapped on a new PE as it cannot be mapped with its master where maximum limit has already reached.

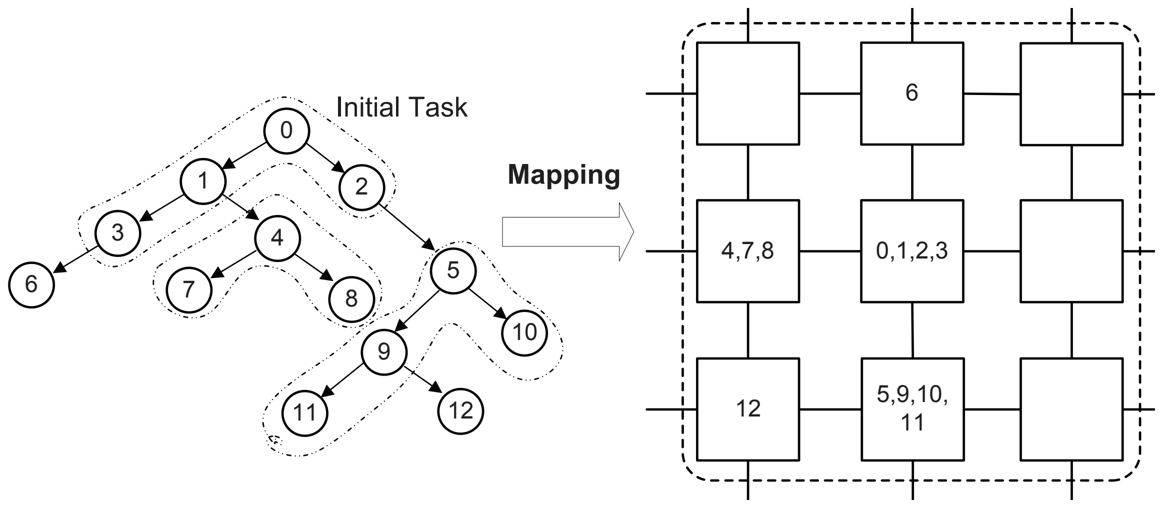


Figure 6.2: An example application mapping by CPBN heuristic

The CPBN technique tries to reduce communication overhead by mapping maximum communicating task pairs on the same PE as highlighted in Figure 6.2. However, it has various drawbacks. First, the technique does not try to balance the computation load on the PEs while reducing communication overhead between tasks. This may lead to unbalanced execution on PEs, resulting in high computation load variance among different PEs. Next, it has a restricted approach rather than a global approach towards minimizing communication overhead as communication can be avoided only between the master and the requested slave tasks during mapping. Further, when the MPSoC contains GPPs and reconfigurable areas (RAs) as PEs, the technique does not fully utilize the capabilities of the RAs by sparsely executing the computation intensive tasks on them. Next, we discuss proposed computation and communication aware mapping strategies that overcome the mentioned drawbacks.

The proposed mapping strategies first perform *pre-processing* of the given applications followed by *mapping* on the MPSoC platform, as shown in Figure 6.3. Here, the *platform manager* is responsible for *pre-processing* too in addition to *mapping*, *scheduling*, *resource control* and *configuration control*. The applications are handled one after another, i.e. one

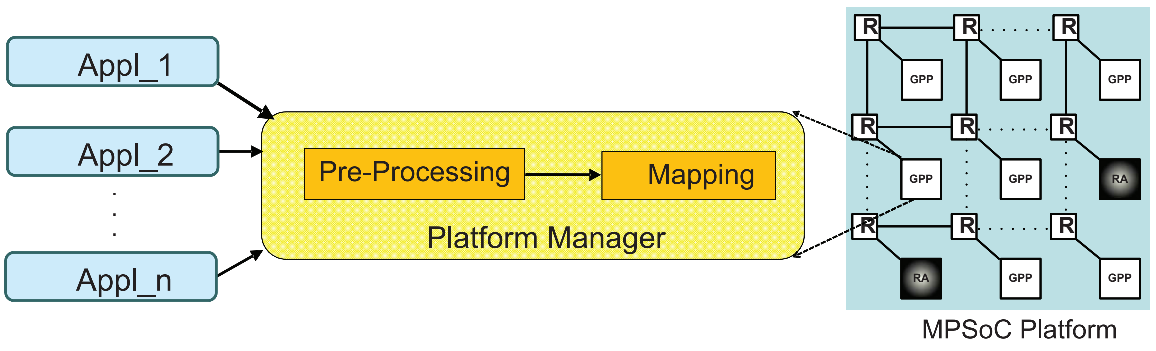


Figure 6.3: Run-time pre-processing and mapping

application is first pre-processed and then mapped before handling next application. Pre-processing is performed based on the known properties of the application and platform, and involves some extra effort for managing computation overhead in addition to the communication overhead.

6.2.1 Pre-processing for Balancing Computation and Communication

In the proposed *pre-processing* techniques, the application task graph is taken as an input and the techniques try to minimize communication latencies among various tasks while simultaneously trying to balance the processing load on various PEs. The techniques start by targeting the communication intensive edges in the application and attempt to merge these highly communicating tasks on the same PE. The merging operation takes place only if memory or area constraint of the involved PE is satisfied, i.e., the PE must have sufficient memory or area to configure for both the tasks and sufficient shared memory for their local communication if required. The proposed strategy forms a global approach as complete application graph is seen in entirety for removing communication bottlenecks. However, the communication-aware mapping techniques proposed in the previous chapter merge tasks on the same PE only when communicating tasks are requested during execution.

The main purpose of pre-processing is to remove any bottleneck that may arise due to overhead of transferring data among communicating tasks. Such bottlenecks can be found by examining the execution trace of the application graph. In Figure 6.1, execution trace of application graph "A" is shown, where r_i represents the time taken by task t_i ($\in T$) to process a packet after it is completely received and w_i represents the time taken to transfer a packet by edge e_i ($\in E$) at full available channel bandwidth. The processing and transfer times are referred to as computation and communication loads, respectively. When task t_i is assigned to a GPP then r_i contains task's software computation load in cycles (t_{swcomp}) and when assigned to a RH then r_i contains the task's hardware computation load in cycles (t_{hwcomp}).

The execution trace clearly expresses the execution pattern of tasks and edges. The trace can be analyzed to remove bottlenecks in order to achieve high performance. For example, edge e_2 taking maximum time to transfer a packet appears to be the main bottleneck in Figure 6.1. If the communicating tasks of edge e_2 (t_1 & t_3) are merged together on a single PE, then e_2 no longer remains the active bottleneck of the system and the whole execution trace will shrink leading to lower overall execution time. The improvements rely on the fact that eliminating the slowest step in a pipeline will certainly lead to performance improvement. Based on eliminating the slowest steps repeatedly, next, we present two pre-processing techniques to be applied when targeting homogeneous and heterogeneous MPSoCs respectively.

6.2.1.1 Pre-processing for Homogeneous MPSoCs

The proposed pre-processing technique for homogeneous MPSoCs is decomposed into two phases and is presented through Algorithm 6. In first phase (step 1 to 12), the strategy examines the application graph and attempts to remove the communication bottlenecks. From the application graph, first, the task having maximum computation load max_p_{load}

Algorithm 6: Pre-processing Technique for Homogeneous MPSoCs

```
Input: TG(T,E)
Output: Optimized TG(T,E)
 1: repeat
 2:    Find node $t_i \in T$ having maximum computation load $max\_p_{load}$ from TG(T,E);
 3:    Find edge $e_i \in E$ having maximum communication load $max\_c_{load}$ from TG(T,E);
 4:    if $max\_p_{load} < max\_c_{load}$ then
 5:       Find computation load of connecting nodes $t_p$ & $t_q$ of the edge $e_i$, i.e., $p_{load}(t_p)$ and $p_{load}(t_q)$;
 6:       if $(p_{load}(t_p) + p_{load}(t_q)) \leq max\_c_{load}$ then
 7:          Merge $t_p$ & $t_q$ to a single node if their memory requirements are satisfied and update
             TG(T,E);
 8:       else
 9:          break; // goto to second phase for optimization, i.e. start from step 13
10:       end if
11:    end if
12: until $max\_c_{load} > max\_p_{load}$
13: repeat
14:    Find communicating nodes $t_i$ and $t_j$ having minimum computation loads $p_{load}(t_i)$ and $p_{load}(t_j)$
       from updated TG(T,E);
15:    if $(p_{load}(t_i) + p_{load}(t_j)) < max\_p_{load}$ in updated TG then
16:       Merge $t_i$ & $t_j$ to a single node if their memory requirements are satisfied and update TG(T,E);
17:    end if
18: until $max\_p_{load} > (p_{load}(t_i) + p_{load}(t_j))$
```

(step 2) and edge having maximum communication load max_c_{load} (step 3) are determined. If the maximum communication load is greater than maximum computation load (step 4) and addition of computation loads of communicating tasks is less than the maximum communication load (step 6), then the communicating tasks are merged on a single PE (step 7). This process is continued till the computation load on any PE becomes the bottleneck and further reduction in communication overhead will not yield any performance improvement.

Once the processor becomes the bottleneck, second phase (step 13 to 18) is carried out for resource optimization. The tasks with minimum computation load are merged such that the communication overhead or computation load does not overshoot the computation bottleneck determined in the first phase. This phase not only enhances resource utilization but also tries to balance the computation load among several PEs by bringing the computation load of each PE as close as possible to the computation bottleneck.

The worst case complexity (C) of the pre-processing strategy has been calculated in terms of number of tasks in the application. For a given number of tasks 'n', the complexity (C) is calculated as follows:

$$C = (n\_choose\_2) + ((n-1)\_choose\_2) + ((n-2)\_choose\_2) + \ldots\ldots + (2\_choose\_2)$$
$$= {}^{n}C_2 + {}^{(n-1)}C_2 + {}^{(n-2)}C_2 + \ldots\ldots + {}^{2}C_2 = \sum_{p=2}^{n} ({}^{n}C_2) = \frac{n^3-n}{6} \qquad \text{(Eq. 6.1)}$$

In the Equation Eq. 6.1, *n_choose_2* specifies the number of ways for choosing two nodes from n nodes. The chosen nodes are merged on the same PE. Similarly, *(n-1)_choose_2* specifies the number of ways for choosing two nodes from $n-1$ nodes where one task is already combined, *(n-2)_choose_2* specifies the number of ways for choosing two nodes from $n-2$ nodes where two tasks are already combined and so on. The worst case complexity is O(n^3) which allows us to preprocess the applications with even large number of tasks in a limited time. However, the preprocessing may converge earlier depending upon the type of application. The preprocessing is not an exhaustive search as tasks mapped on a node are not split up for the next iteration of node merging. On the other hand in an exhaustive search, splitting of the merged nodes is required in order to evaluate all the combinations exhaustively.

For an example demonstration, consider the pre-processing of the same application graph as in Figure 6.2. The computation and communication loads were not shown in Figure 6.2 as the CPBN strategy does not need loads' information. Figure 6.4 shows the same application as 'initial application' with computation and communication load values in cycles when tasks are executed on GPPs and edges are allocated full communication (channel) bandwidth. Initially, the edge between task 4 and 8 is the main bottleneck (maximum communication load) and computation load of task 4 & task 8 is less than the communication load, so task 4 and task 8 are merged together. Next, the edge between

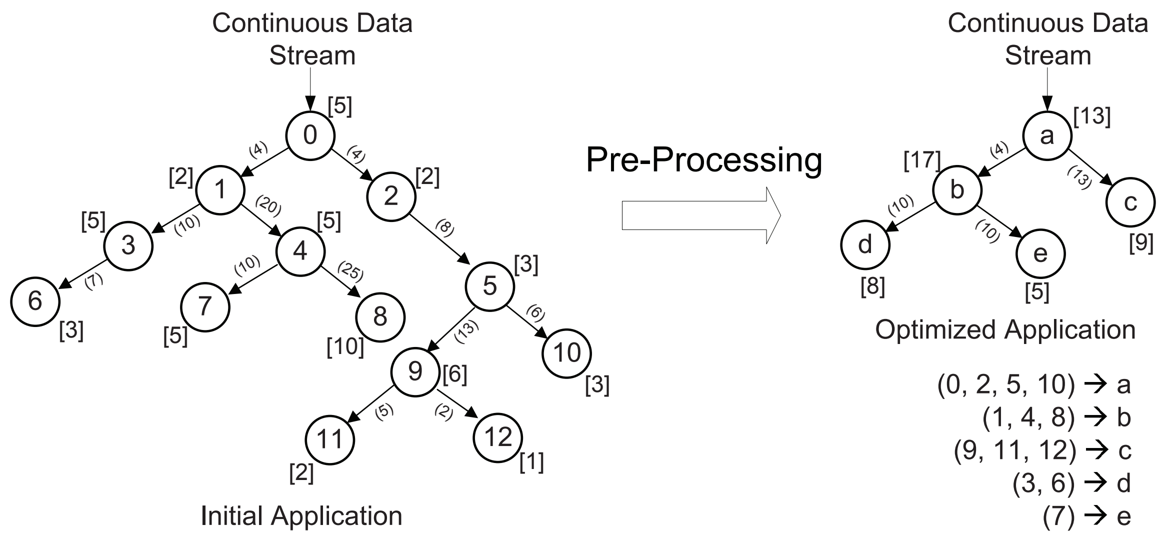


Figure 6.4: Pre-processing to get optimized application by Algorithm 6

task 1 and 4 becomes the bottleneck (maximum communication load in the updated TG) and the sum of computation load of task 1 and task 4 is less than the communication load, so task 1 and task 4 are merged together. This process is repeated till computation load on any GPP becomes the bottleneck, i.e. maximum computation load on a GPP becomes greater than maximum communication load. The GPP bottleneck arises when tasks 1, 4 and 8 are merged on the same PE. Thereafter, resource optimization is carried out first by merging communicating tasks 2 and 5 as they have minimum sum for their computation loads. This process is also repeated till an optimized graph is obtained where further resource optimization cannot be done without overshooting the computation bottleneck.

The output of the pre-processing operation is an optimized application graph named 'Optimized Application' as shown in Figure 6.4. This graph contains different set of tasks at each node, like, tasks 0, 2, 5 and 10 on node a, tasks 1, 4 and 8 on node b, tasks 9, 11 and 12 on node c, tasks 3 and 6 on node d and task 7 on node e.

Algorithm 7: Pre-processing Technique for Heterogeneous MPSoCs

```
Input: TG(T,E)
Output: Optimized TG(T,E)
1: Repeat steps 1 to 18 of Algorithm 6 by considering hardware computation load $p_{hw\_load}$ in place of
   computation load $p_{load}$ and area requirement in place of memory requirement; // First phase
   (steps 1 to 12) for removing communication bottlenecks; Second phase (steps 13 to 18) for resource
   optimization
2: for all node $t_i$ in the updated graph do
3:    $\Sigma$ = Sum of software computation load $p_{sw\_load}$ of individual merged tasks on node $t_i$;
4:    if $\Sigma < max\_p_{hw\_load}$ then
5:       Assign $t_i$ to a software resource having computation load equal to $\Sigma$ and update TG(T,E);
6:    end if
7: end for
```

6.2.1.2 Pre-processing for Heterogeneous MPSoCs

The proposed pre-processing technique for heterogeneous MPSoCs is composed of three phases and is presented through Algorithm 7. The first two phases are similar to those of Algorithm 6, where, communication bottlenecks are removed in the first phase and resource optimization is carried out in the second phase. The technique assumes that each task can be executed on both hardware and software PEs. The algorithm starts by examining the application graph when all the tasks are assigned to hardware resources, i.e. RAs and hence it starts optimizing the application graph by considering hardware computation load of each task. Therefore, in the first and second phase of Algorithm 7, the same steps as of the Algorithm 6 are repeated by considering hardware computation load p_{hw_load} in place of computation load p_{load} and checking for area requirements in place of memory requirements while merging on a single RA node.

After first and second phase, the resulted optimized application graph contains hardware computation bottleneck as the graph has been obtained by considering hardware computation load of each task. The limited number of hardware resources in the platform restricts the possibility of mapping the optimized graph onto the platform. Therefore, in the third phase of Algorithm 7 (step 2 to 7); we confine the usage of hardware resources depending upon the hardware computation bottleneck $max\_p_{hw\_load}$. The nodes of the optimized graph are analyzed and considered to be executed on software resources if the total software computation time of tasks on the node does not exceed the hardware computation bottleneck obtained earlier. The resultant graph is a combination of hardware and software nodes that need to be executed on the platform. In the graph, computation intensive tasks are assigned to RAs in order to provide improved performance.

For an example demonstration, consider pre-processing of an application graph named 'Initial Application' as shown in Figure 6.5. The application is shown with its tasks' and edges' indicating their computation and communication load values in cycles, respectively. For each task, computation load values are shown when the task is executed on both the hardware (RA) and software (GPP) as $[t_{hwcomp}, t_{swcomp}]$. After pre-processing by applying Algorithm 7, we get an optimized application graph named 'Optimized Application' as shown in Figure 6.5. This graph contains set of tasks at each node, like, tasks 0, 2, 4 and 7 on node a, tasks 1 and 3 on node b, tasks 5 and 8 on node c, task 6 on node d and task 9 on node e. The third phase of the algorithm determines that nodes a and c need to be mapped on hardware resources, whereas the remaining nodes on software resources as shown in the figure.

6.2.2 Mapping of the Processed Application

The optimized application graph thus obtained by the pre-processing can be mapped on the MPSoC by using any of the mapping algorithms proposed in the previous chapters. For demonstration, mapping algorithm PBN proposed in Chapter 4 is applied on the optimized application graph of Figure 6.5 in order to map it on MPSoC. Figure 6.6 shows mapping of the optimized application on part of the MPSoC. First, initial node a is mapped and its slave nodes b, c and d are requested to be mapped. Their placement is found on nearest possible PE with respect to the master node a. After mapping nodes b, c and d, slave node e is requested and mapped. A possible mapping for all the nodes is shown in the Figure 6.6.

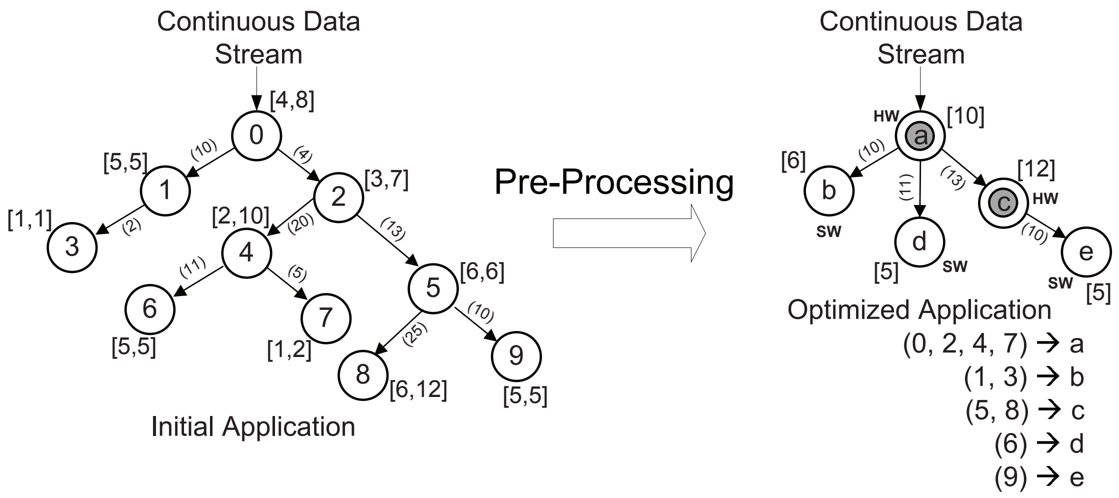


Figure 6.5: Pre-processing to get optimized application by Algorithm 7

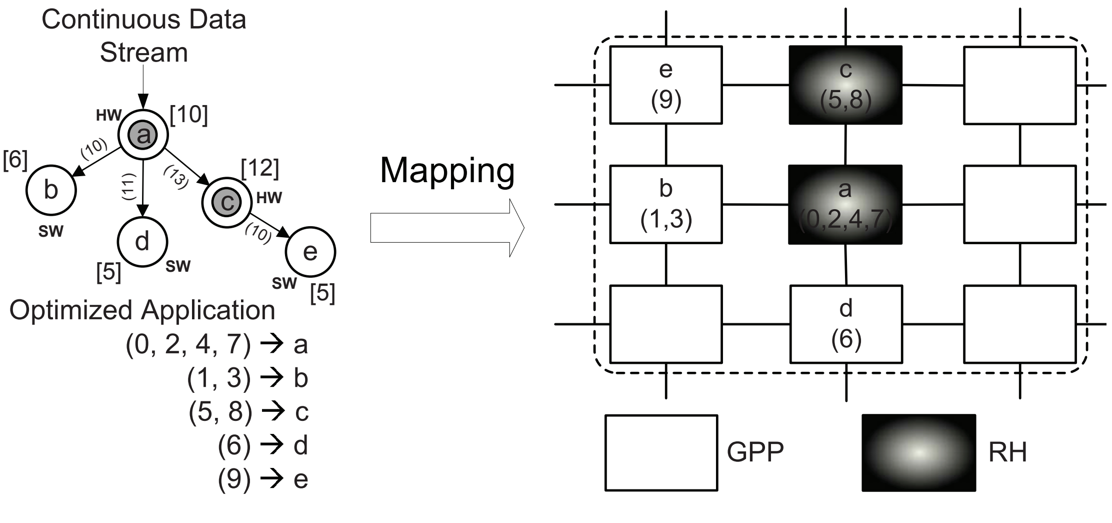


Figure 6.6: Mapping of optimized application

6.3 Performance Evaluation

In this section, we evaluate performance provided by the proposed computation and communication aware mapping strategies. The evaluated performance metrics are *total execution time*, *energy consumption*, *resource utilization* and *computation load variance.*

The simulation platform used to perform the experiments is the same as in Chapter 4, where each processing node supports more than one task. We have performed simulation by assuming maximum four tasks per node in order to avoid the memory space and reconfigurable area availability issues.

Applications are modeled to describe data stream processing behavior with tasks and edges as described in Section 6.1. Each task is considered to be executable on both software and hardware resources, and edges are allocated full channel bandwidth. The simulation is performed by varying the number of streams to be processed by the application.

The evaluated scenarios are:

(i) multiple random, pipeline & tree like streaming applications having 5, 10, 15 and 20 tasks.

(ii) 20 similar MPEG-4 application, where 5 instances are run concurrently.

Evaluations are performed by employing both the pre-processing techniques targeting for homogeneous and heterogeneous MPSoCs. NoC-based 2D-mesh architectures have been considered. In the homogeneous MPSoC, all the PEs are GPPs and in the heterogeneous MPSoC, 24 PEs are GPPs and rest are RHs. In both the MPSoCs, one PE is used as manager processor. Similar to previous chapter, the PEs reserved for the initial task are pre-defined.

The communication-aware packing-based best neighbor (CPBN) mapping technique proposed in the previous chapter and those proposed in this chapter are evaluated and compared. The CPBN technique optimizes for performance metrics energy consumption, channel load and packet latency by a maximum amount with some improvement in total execution time too. We have considered CPBN as the reference technique to compare with the techniques proposed in this chapter. When employing pre-processing for homogeneous and heterogeneous MPSoCs, the mapping strategies are referred to as preprocessing-based homogeneous (PHomog) and preprocessing-based heterogeneous (PHeterog) strategy, respectively.

6.3.1 Total Execution Time

The total execution time is the overall duration to execute the applications for a defined number of streams. It includes pre-processing, mapping, configuration, computation and communication time. The communication overhead stands out as the predominant bottleneck, which is greatly reduced by our proposed strategy. Thus, overall execution time gets reduced.

Figure 6.7 (a) shows overall execution time for the first simulated scenario when mapping heuristics CPBN and PHomog are employed. Figure 6.8 (a) shows the same when heuristics CPBN and PHeterog are employed. In Figure 6.7, result is for homogeneous architectures, whereas in Figure 6.8, it is for heterogeneous architectures. The figures show that our heuristics PHomog and PHeterog perform better than the communication-aware heuristic CPBN at varying number of tasks in the considered applications. It is evident that rise in complexity of the application in terms of number of tasks corresponds to rise in the improvement in execution time with our techniques.

6.3.2 Energy Consumption

Energy is consumed while transmitting a packet from source PE to destination PE and then in processing the packet at the destination PE once it is received. The same energy

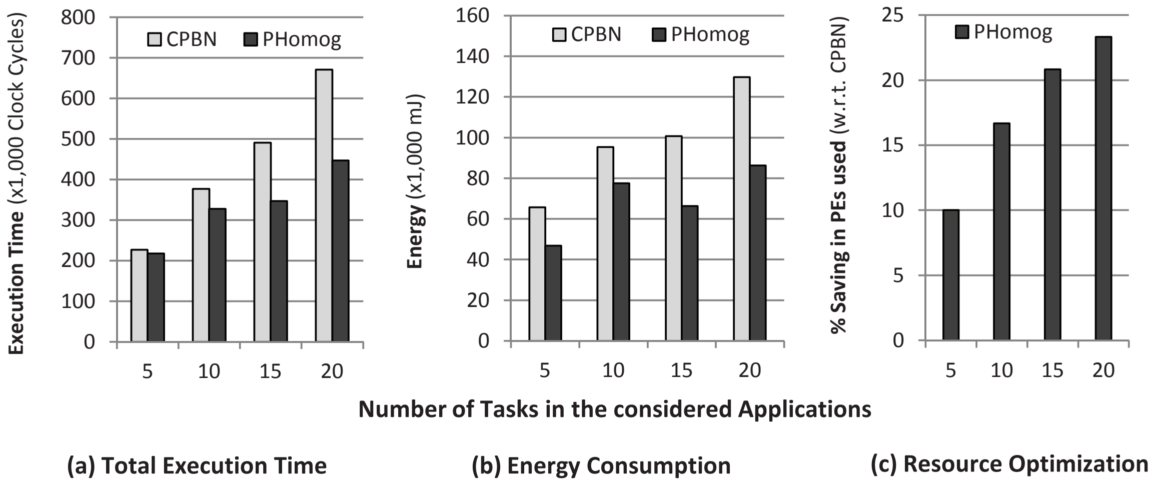


Figure 6.7: Total execution time, energy consumption and resource utilization when CPBN and PHomog heuristics are employed

consumption model as proposed in the previous chapter has been adapted, where energy required in transmission and processing are referred to as communication and computation energy, respectively. For packet transmission, full channel bandwidth has been allocated but it may be varied.

Figure 6.7 (b) shows energy consumption for the first simulated scenario when mapping heuristics CPBN and PHomog are employed. Figure 6.8 (b) shows the same when heuristics CPBN and PHeterog are employed. It can be seen that our approaches PHomog and PHeterog reduce energy consumption as they aim to remove the bottlenecks involved in communication intensive edges, which reduces communication energy. Further, reduction in energy consumption is more as the complexity of application (number of tasks) increases.

6.3.3 Resource Optimization

The resource optimization is measured as the percentage decrease (saving) in the number of PEs (hardware or software) utilized by our approaches over the existing approaches. Figure 6.7 (c) shows software resources savings by our PHomog mapping technique over

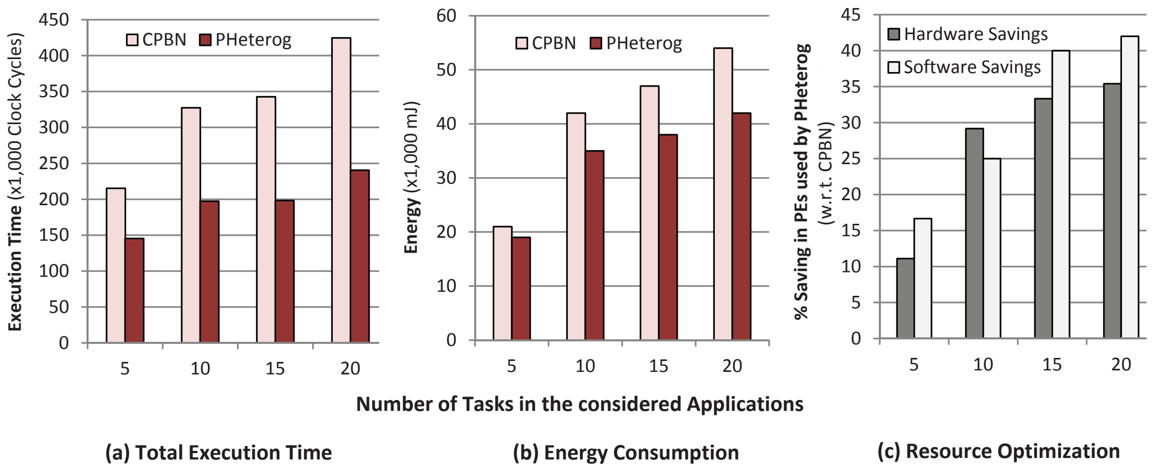


Figure 6.8: Total execution time, energy consumption and resource utilization when CPBN and PHeterog heuristics are employed

the CPBN technique for applications having different number of tasks. Figure 6.8 (c) shows hardware and software resources savings by our PHeterog technique over the CPBN technique.

Our techniques reduce the number of hardware and software resources used by the application as they try to map maximum number of tasks on the same PE while reducing communication overhead and balancing the processing load among various PEs. Therefore, our techniques provide resource savings. Further, it can be observed that resource savings get increased as applications having larger number of tasks are considered. For applications with 20 tasks, PHomog shows an improvement of 23.33%, as shown in Figure 6.7 (c). For applications with 15 tasks, PHeterog shows an improvement of 33.33% and 40.00% for hardware and software resources respectively, as shown in Figure 6.8 (c).

6.3.4 Computation Load Variance

The computation load variance is measured as the standard deviation of computation loads on different PEs. It determines how uniformly computation loads are distributed on the PEs. Lower value of standard deviation shows better uniform distribution of loads

		Computation Load (in cylcles) on different Software (GPP) PEs						Standard Deviation
		GPP 1	GPP 2	GPP 3	GPP 4	GPP 5	GPP 6	
10 Tasks	CPBN	14	21	3	19	6	NA	7.0597
	PHomog	14	17	18	7	5	NA	5.26
15 Tasks	CPBN	2	10	11	15	13	1	5.3124
	PHomog	13	17	8	5	9	NA	4.176

Table 6.1: Computation load distribution and their variance for applications with 10 and 15 tasks when heuristics CPBN and PHomog are employed.

on the PEs. Fair distribution of computation load among the several PEs minimizes the probability of reaching the situation when a single PE remains active for most of the time due to high computation load, whereas other PEs remain idle. This scenario leads to poor power efficiency and performance bottlenecks due to overloading of a single PE. Our proposed heuristics attempt to fairly distribute the computation load among several PEs by combining the less computation intensive tasks onto a single PE till performance bottleneck is detected.

Table 6.1 shows distribution of computation load among several PEs and its standard deviation for applications in the first simulation scenario when CPBN and PHomog heuristics are employed. The GPPs are used as PEs. The distribution and its standard deviation clearly indicate that our heuristic PHomog provides more uniform distribution and thus smaller deviation from the mean computation load.

Table 6.2 shows the distribution of computation load among hardware and software PEs when CPBN and PHeterog heuristics are employed. When employing CPBN, the distribution indicates that hardware resources are lightly loaded along with highly uneven distribution of loads among PEs. Our heuristic PHeterog provides efficient utilization of hardware and software resources and better distribution of loads when compared to CPBN.

		Computation Load (in cylcles) on different Hardware (RH)/Software (GPP) PEs								Standard Deviation
		RH 1	RH 2	RH 3	GPP 1	GPP 2	GPP 3	GPP 4	GPP 5	
10 Tasks	CPBN	3	3	5	14	6	1	11	NA	4.35655
	PHetero	11	12	NA	6	5	5	NA	NA	3.42053
15 Tasks	CPBN	1	5	1	15	9	3	4	2	4.5
	PHetero	12	12	NA	4	5	8	NA	NA	3.76829

Table 6.2: Computation load distribution and their variance for applications with 10 and 15 tasks when heuristics CPBN and PHeterog are employed.

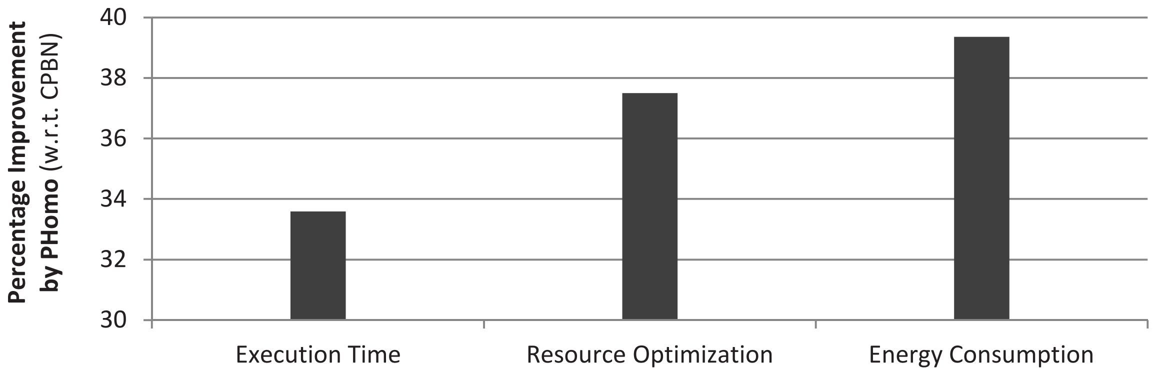


Figure 6.9: MPEG4 decoder application case study

6.3.5 Case Study - MPEG4 Decoder

The proposed heuristics have been applied on real-life MPEG-4 decoder applications as mentioned in scenario (ii). The MPEG4-decoder is used in de-compression of encoded video digital data. It has been modeled as a task graph composed of 13 tasks interconnected with each other in a cyclic tree like structure. Figure 6.9 shows improvement in execution time, resource utilization and energy consumption by our proposed technique PHomog when compared to CPBN. The proposed technique reduces total execution time by 33.59%, resource utilization by 37.5% and energy consumption by 39.35%.

6.4 Summary

In this chapter, we have proposed novel computation and communication aware mapping strategies, where placement for a task is found aiming at balancing the computation load on different PEs and reducing communication overhead in the MPSoC platform. The strategies pre-process the application before actual mapping is done. In pre-processing, communication bottlenecks are removed while balancing computation load on different PEs at the same time. The pre-processing also performs resource optimization by merging the tasks on the same PE if performance is not degraded and memory or area requirements on the PE are satisfied. The preprocessed application is then mapped on the platform PEs.

Based on our investigations to map models of real-life streaming applications on MP-SoC platforms, we have shown that the proposed mapping strategies outperform existing mapping strategies. We have evaluated total execution time, energy consumption, resource optimization and computation load variance for different application scenarios. The improvements in different scenarios are clearly enunciated in performance evaluation.

The run-time mapping strategies proposed till now (from Chapter 3 to this chapter) perform all the processing at run-time, i.e. on-the-fly processing. These strategies cannot guarantee for strict timing deadlines due to limited compute resources at run-time. In the next chapter, we present a hybrid mapping strategy that performs compute intensive analysis at design-time and uses the analysis results at run-time in order to accelerate the run-time mapping process towards meeting the deadlines.

Chapter 7

Hybrid Strategy for Accelerating Run-time Mapping

In the previous chapters (Chapter 3 to Chapter 6), we have presented run-time mapping strategies where all the processing is performed at run-time in order to cater for dynamism in applications. However, these strategies cannot always guarantee for schedulability, i.e., for strict timing deadlines due to lack of any prior analysis and limited compute resources at run-time. Thus, there is a need to devise a strategy that should perform compute intensive analysis at design-time and should use the analyzed results at run-time to overcome the above mentioned problem.

In this chapter, we present a hybrid approach for efficient run-time mapping of applications on MPSoCs. For each application to be supported in the MPSoC, the approach performs extensive design-space exploration (DSE) at design-time to derive multiple design points representing throughput and energy consumption at different resource combinations. One of these points is selected at run-time efficiently depending upon the desired throughput while optimizing for the energy consumption and resource usage. While most of the existing DSE strategies consider a fixed multiprocessor platform architecture, our DSE considers a *generic* architecture making DSE results applicable to a large set of target platforms. Further, existing DSE strategies do not scale well with the application

& platform size and do not always provide the largest throughput design points. Experimental results reveal that the proposed approach provides faster DSE, better design points and efficient run-time mapping when compared to other approaches. In particular, we show that DSE is faster by 83% and run-time mapping is accelerated by 93% for some cases. Part of the work in this chapter has been published in [C-9] and [C-11]. Extended versions of the work has been accepted to be published in [J-2] and [J-3].

The remainder of the chapter is organized as follows. Section 7.1 introduces some preliminaries necessary to understand the hybrid mapping strategy. The hybrid mapping flow that first performs design-time analysis of applications and then map the applications on a multiprocessor platform at run-time is presented in Section 7.2. In Section 7.3, we demonstrate implementation of the hybrid strategy with some example applications. The performance of the strategy is evaluated in Section 7.4. Section 7.5 summarizes the chapter.

7.1 Preliminaries

This section covers some preliminaries necessary to explain our proposed hybrid mapping flow. We describe the target platform and application model with the underlying assumptions and terminology.

Multiprocessor Platform Model

The multiprocessor platform model used in this work is similar to that of previous chapters. The platform contains a set of tiles and an interconnection network to connect the tiles. Here, end-to-end connections (c) with fixed latency between tiles are used to connect the tiles. The connections of the network can be reserved in advance to avoid resource contention and to provide guaranteed throughput. In contrast, in a packet switched network as used in the previous chapters, providing throughput guarantee is

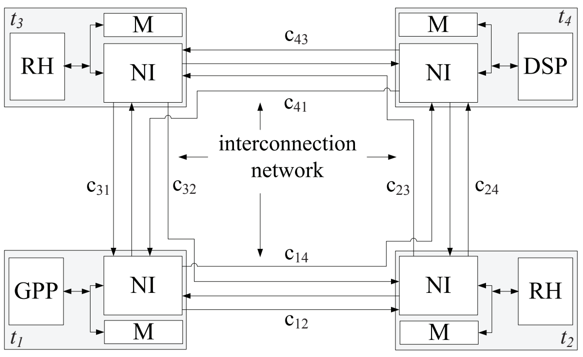


Figure 7.1: Example multiprocessor platform.

tile	p_{type}	*m*	*ci*	*co*	*iω*	*oω*	*pwr*	connection	*L*
t_1	GPP	1000000	8	8	12	12	1000	c_{12}, c_{24}, c_{43}, c_{31}	3
t_2	RH	1000000	8	8	12	12	4.60	c_{14}, c_{23}, c_{41}, c_{32}	6
t_3	RH	1000000	8	8	12	12	4.60		
t_4	DSP	1000000	8	8	12	12	330		

Table 7.1: Properties of the example platform.

difficult since there are no dedicated connections [59] [159] [64]. Figure 7.1 shows an example platform, where end-to-end connections are used to connect the tiles t_1, t_2, t_3 & t_4. Each connection may have a different latency, so the latency of connections through a network-on-chip (NoC) can be taken into account [160], i.e., any type of interconnection network can be modeled so long the latencies between tiles are provided. Similar to previous chapters, each tile contains a processor (for example, general purpose processor (GPP), digital signal processor (DSP) or reconfigurable hardware (RH) as shown in Figure 7.1), a local memory (M) and a network interface (NI) containing set of communication buffers that are accessed both by the interconnect and the local processor. Next, we provide a formal definition of the multiprocessor platform.

Definition 7.1 (Platform Graph (PG)) *A PG is represented as (T,C,L), which contains a set T of tiles, a set C of connections and a latency function L that provides latency (in time-units) of each connection (L(c)). A tile* $t \in T$ *is a 7-tuple* $(p_{type}, m, ci, co, i\omega, o\omega, pwr)$*,* *where, $p_{type} \in PT$ (PT is set of processor types), m is the memory size (in bits), ci & co are the maximum number of input and output connections supported by the NI, iω & oω are the maximum incoming and outgoing bandwidth (in bits/time-unit) and pwr is the power consumption (in milliwatts) of the processor type p_{type}..*

Table 7.1 shows the values of all the elements in the example platform graph (Figure 7.1). Multiprocessor systems such as StepNP [34], PROPHID [35] and Eclipse [36] fit nicely into this platform model.

Similar to previous chapters, the communication network in Figure 7.1 is arranged in a 2D-mesh topology and distance between two tiles is referred to as hop_distance. The latencies of end-to-end connections are modeled according to the 2D-mesh network. Tiles t_1 & t_2 are at hop distance of 1 (just adjacent) and t_1 & t_4 at hop distance of 2 as communications are via tile t_2 (1 hop in X-direction to reach tile t_2 and then 1 hop in Y-direction to reach tile t_4). The hop_distance between the tiles determines the latency of the connections that connect the tiles. The application edges are mapped onto the connections between tiles. Each edge occupies one connection between the tiles at its full bandwidth and the occupied connection always serves only to the assigned edge in order to provide through guarantee. Unoccupied connections can be used for other edges when required. The latency is directly proportional to the hop_distance that determines the length of the connection. Table 7.1 shows the latencies of connections according to the hop_distance. To incorporate that two tiles are at higher hops, we change the latency of the connections between the tiles according to the hops. This incorporation helps in finding mappings when the tiles are further apart in the actual platform.

Application Model

The models of streaming applications described in the previous chapter do not consider timing constraints. However, modern embedded systems need to support time-constrained streaming multimedia applications. Therefore, application models having

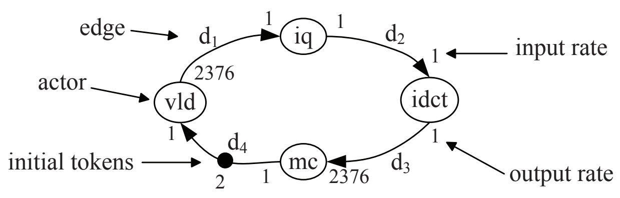


Figure 7.2: SDFG model of an H.263 decoder.

actors	p_{types}	GPP(ET,mem)	ACC(ET,mem)	RH(ET,mem)	edges	sz	$mreq_t$	$mreq_{src}$	$mreq_{dst}$	ω
vld	GPP,ACC	(26018,10848)	(13009,10848)	(–,–)	d_1	512	2376	2376	1	5
iq	GPP,ACC	(559,400)	(450,400)	(–,–)	d_2	512	1	1	1	5
idct	GPP	(486,400)	(–,–)	(–,–)	d_3	512	2376	1	2376	5
mc	GPP,RH	(10958,8000)	(–,–)	(5479,8000)	d_4	1216512	3	1	1	5

Table 7.2: Resource requirement of actors and edges of H.263 decoder.

timing constraints need to be considered to realize realistic scenarios. Here, Synchronous Dataflow Graphs (SDFGs) [161] are used to model concurrent multimedia applications with timing constraints as they facilitate for easier modeling [93]. The SDFG model of H.263 decoder is shown in Figure 7.2. The nodes model the tasks and are referred to as *actors*, which communicate with *tokens* sent from one actor to another through the edges modeling dependencies. The H.263 decoder is modeled with four actors *vld, iq, idct* & *mc* and four edges *d1, d2, d3* & *d4*. An actor *fires* (executes) when there are sufficient input tokens on all of its input edges and sufficient buffer space on all of its output channels. Every time an actor *fires*, it consumes a fixed amount of tokens from the input edges and produces a fixed amount of tokens on the output edges. These token amounts are referred to as *rates.* The rates determine how often actors have to fire with respect to each other. The edges may contain *initial tokens*, which are indicated by a bullet point, as in Figure 7.2.

Definition 7.2 (SDFG) *An SDFG (A,E) consists of a set A of actors and a set E of edges. An edge* $e = (a_1,a_2,tk_1,tk_2)$ *represents a dependency of actor* a_2 *on* a_1*. When* a_1 *fires, it generates* tk_1 *tokens on e and when* a_2 *fires, it consumes* tk_2 *tokens from e. Initial tokens on edges are defined as* $TokIn : E \rightarrow$ *natural numbers including 0.*

Analysis techniques to calculate throughput and storage requirements for an SDFG already exist [162]. Throughput is an important constraint for multimedia applications. Throughput is defined as the inverse of the long term period, i.e., the average time needed for one iteration of the application. Iteration is defined as the minimum non-zero execution such that the original state of the SDFG is obtained. For example, in Figure 7.2, period of H.263 decoder is = ExecTime(*vld*) + 2376.ExecTime(*iq*) + 2376.ExecTime(*idct*) + ExecTime(*vld*), where ExecTime is execution time. It should be noted that actors *iq* and *idct* have to execute 2376 times. This period is just for demonstration and does not include network and memory access delays. An SDFG with a throughput of 100 Hz takes 10 ms to complete one iteration.

For modeling an application, resource requirements of the actors and edges are clearly specified. The application model also specifies a throughput-constraint that must be satisfied when the application is mapped onto the platform.

Definition 7.3 (Application Graph (AG)) *An AG is represented as (A,E,AP,EP) which is derived from SDFG (A,E). AP and EP provide resource requirement of actors and edges on the platform, respectively. For each actor $a \in A$, AP provides a tuple (ET,mem) for each implementation alternative ($\in p_{types}$), where, p_{types} represents the implementation alternatives of the actor, ET and mem represent the execution time (in time-units) and memory needed (in bits) on the implementation alternative, respectively. AP provides null values for ET and mem for unsupported implementation alternatives. For each edge $e = (a_1,a_2,tk_1,tk_2) \in E$, EP provides a 5-tuple $(sz,mreq_t,mreq_{src},mreq_{dst},\omega)$, where, sz is size of a token (in bits), $mreq_t$ is the memory (in tokens) needed when a_1 and a_2 are allocated to the same tile, $mreq_{src}$ and $mreq_{dst}$ is the memory (in tokens) needed in source and destination tile respectively and ω is the bandwidth (in bits/time-unit) needed when a_1 and a_2 are allocated to different tiles. The throughput constraint of the AG is represented as τ.*

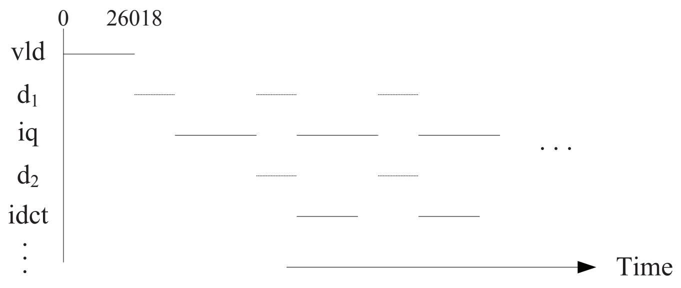


Figure 7.3: Execution trace of H.263 decoder.

Table 7.2 represents the values of *AP* and *EP* for actors and edges of the H.263 decoder application. In the previous chapter, it was assumed that each actor have its implementation alternatives as GPP & RH and each edge to be assigned full connection bandwidth between the tiles. Here, implementation alternatives and required bandwidth are varied. Execution pattern of the H.263 decoder (consisting of 4 actors) mapped on a 4-tile MPSoC platform such that each actor is mapped on a different GPP (ARM processor) tile is shown in Figure 7.3. It is clearly seen that actors *iq* and *idct* have potential to execute in parallel. It has been observed that when the existing strategies are applied to perform design-time analysis for the H.263 decoder on a 3-tile platform, in some cases, the best produced mapping contains actors *iq* and *idct* on the same tile while optimizing for some performance metrics such as power and resource optimization. For example, the strategy in [138] maps actors *iq* and *idct* on the same tile while optimizing for load balancing on the three used tiles for the application. This forces their execution sequentially, resulting in reduced throughput. However, we will show that our approach finds the best mapping which has the maximum throughput where actor *iq* and *idct* are not allocated on the same tile, but sequentially executing actors like *vld* and *iq* on the same tile. Mapping the connected and sequentially executing actors on the same tile results in reduced communication overhead between the actors, which may maximize the throughput even when using smaller number of tiles.

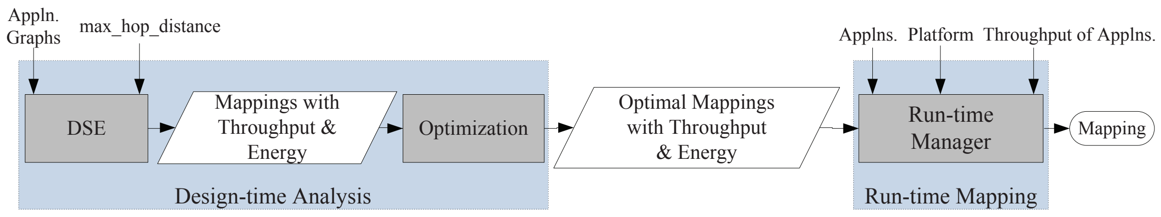


Figure 7.4: Hybrid mapping strategy.

7.2 Hybrid Mapping Strategy

This section details our hybrid mapping strategy. The strategy is presented in Figure 7.4. It has two main steps: *1)* analysis of applications at design-time (*Design-time Analysis*), and *2)* mapping of the applications on a platform by using the analysis results (*Optimal Mappings with Throughput & Energy*) with the help of a platform manager (*Run-time Manager*) at run-time.

7.2.1 Design-time Analysis

The *Design-time Analysis* step evaluates a number of mappings for each application to be supported onto a hardware platform. The applications are evaluated one after another. The evaluation considers finding different mappings and their throughput & energy consumption. For each mapping, actors (A) and edges (E) of the application graph AG are bound to tiles (T) and connections (C) between two tiles or the memory inside a tile in the platform graph PG. This binding gives a resource allocation for the application graph AG on the platform graph PG with the following constraints for each tile $t \in T$:

(i) (memory imposed by actors and edges bound on t) $\leq$ (memory (m) on t),

(ii) (allocated input connections on t) $\leq$ (maximum input connections ci on t),

(iii) (allocated output connections on t) $\leq$ (maximum output connections co on t),

(iv) (allocated incoming bandwidth on t) $\leq$ (maximum incoming bandwidth $i\omega$ on t),

(v) (allocated outgoing bandwidth on t) $\leq$ (maximum outgoing bandwidth $o\omega$ on t).

Throughput & Energy Consumption Computation

The throughput for a mapping is computed by taking the resource allocations into account. First, static-order schedule for each tile is constructed that orders the execution of bound actors. A list-scheduler is used to construct the static-order schedules for all the tiles at once. Then, all the binding and scheduling decisions are modeled in a graph called binding-aware SDFG. Finally, throughput is computed by self-timed state-space exploration of the binding-aware SDFG [162].

The energy consumption for a mapping is computed as sum of the communication and computation energy for all the tasks for one iteration of the application. Communication energy is required to transfer data from source tile to destination tile through a connection when actors mapped on the two tiles need to communicate with each other. The communication energy is estimated as product of the number of bits to be transferred, number of hops to be traversed between the two tiles and energy required to transfer one bit through one hop, for each edge (e) mapped to a connection (conn) from equation Eq. 7.1. The transferred bits through a connection are calculated as the product of the number of tokens to be transferred and the token size for the edge mapped on the connection. The number of tokens for an edge (e) is computed as the product of repetition vector of source (or destination) actor and source (or destination) port rate of the edge from equation Eq. 7.2. The energy required to transfer one bit through one hop is denoted as E_{Lbit} [156] [157]. Computation energy is required to process the transferred token on the destination tile after it is received and able to fire (execute) the mapped actor. The computation energy for each actor (a) mapped to tile (t) is estimated as product of the number of executions of actor a, execution time and power consumption of a on tile t from equation Eq. 7.3. Total energy consumption is measured as sum of communication and computation energy.

$$E_{comm} = \sum[(e \rightarrow nrTokens) \times (e \rightarrow tokenSize) \times hopCount \times E_{Lbit}] \qquad \text{(Eq. 7.1)}$$

$$e \rightarrow nrTokens = repVector[e \rightarrow srcActor] \times (e \rightarrow srcPortRate) \qquad \text{(Eq. 7.2)}$$

$$E_{comp} = \sum[repVector[a] \times (a \rightarrow execTime(t \rightarrow procType)) \times procPower] \qquad \text{(Eq. 7.3)}$$

The *Design-time Analysis* for an application (*Appln. Graph*) first performs design space exploration (*DSE*) to obtain design points that contain mappings and their corresponding throughput and energy consumption (*Mappings with Throughput* & *Energy*). Then, an *optimization* on the explored design points to get only the Pareto-optimal design points (*Optimal Mappings with Throughput* & *Energy*) providing through and energy consumption at different resource combinations (Fig. 7.4). The DSE flow is presented in Fig. 7.5.

The DSE flow first considers a *generic* platform graph (*T,C,L*) that can cover all the possible mappings for the application graph (*A,E,AP,EP*) to be analyzed currently. A platform containing n tiles of each implementation alternative provided in the application is considered, where n is the number of actors in the application. This platform is capable of covering all the potential mappings. Considering any bigger platform wouldn't provide better performance as the considered one can exploit all the parallelism present in the application. However, all the parallelism might not be exploited if a small size platform is considered where concurrent executing tasks will get mapped on the same tile.

Initially, the considered platform contains tiles with separation between them as one hop_distance (*hop_distance = 1*), which provides a minimum fixed latency for all the

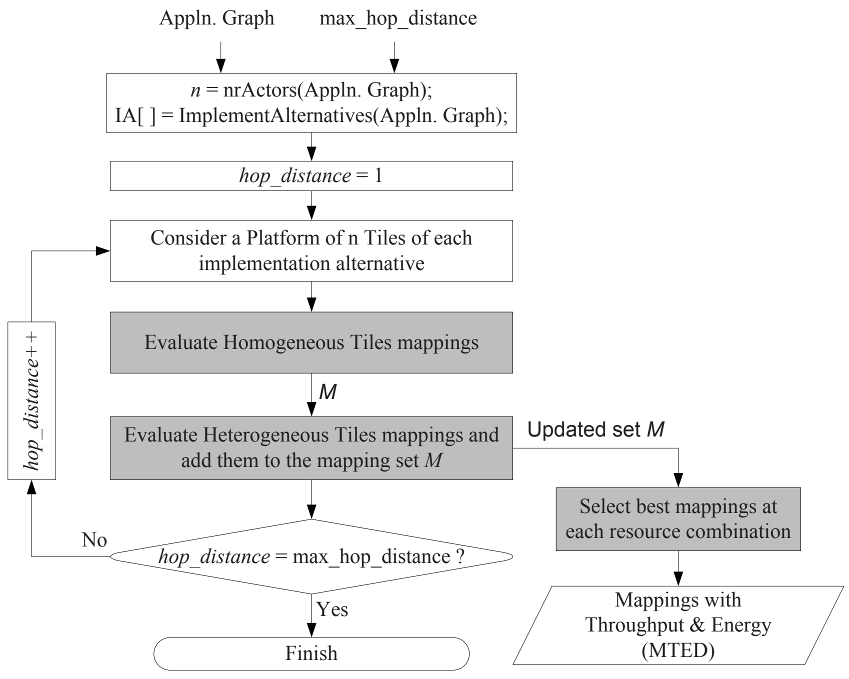

Appln. Graph
max_hop_distance
n = nrActors(Appln. Graph);
IA[ ] = ImplementAlternatives(Appln. Graph);
hop_distance = 1
Consider a Platform of n Tiles of each implementation alternative
Evaluate Homogeneous Tiles mappings
M
Evaluate Heterogeneous Tiles mappings and add them to the mapping set M
Updated set M
Select best mappings at each resource combination
Mappings with Throughput & Energy (MTED)
hop_distance++
No
hop_distance = max_hop_distance ?
Yes
Finish

Figure 7.5: Design-time DSE flow.

connections between the tiles. The DSE flow is repeated by considering a similar platform that contains tiles with separation of one higher hop_distance (*hop_distance++*) between them, i.e., with increased latency for connections, till the *hop_distance* reaches to *max_hop_distance* (one of the input to the DSE flow). The designers can choose an appropriate value of *max_hop_distance* depending upon the expected hardware platform at run-time, where, maximum separation between the tiles can be up to *max_hop_distance*. For a higher value of *max_hop_distance*, the design-time DSE evaluates larger number of mappings. This requires more evaluation time but the applicability of mappings get increased. For example, evaluated mappings with *max_hop_distance* value of 6 are applicable to any platform where maximum separation between the tiles is less than or equal to 6 hops such as mesh of 2×2, 2×3, 3×3 and 4×4 tiles platforms.

By considering varying values of hop_distance, we get mappings where each edge of the application is mapped to a connection at hop_distance of one (to account for minimum latency) to *max_hop_distance* (to account for maximum latency). This facilitates us to cater for the run-time aspects when the available tiles are at different hop_distances. A strategy to find the best mapping in such run-time scenarios is presented in Section 7.2.2. We have considered generic tile architecture so any type of interconnection network can be modeled. The main steps of the DSE flow (highlighted in Fig. 7.5) and *optimization* technique are described subsequently.

7.2.1.1 Evaluating Homogeneous Tiles Mappings

This step evaluates mappings using only GPP tiles. The mappings can be explored in different ways. One way could be exhaustive design space exploration where all the possible actors to tiles combinations, i.e. mappings are evaluated. However, exhaustive exploration is not scalable as the number of combinations grows exponentially with the number of actors or tiles. Thus, it may take days or weeks for large application and/or platform size. The application and platform size are referred to as the number of actors and tiles, respectively. To overcome the large exploration overhead, we have devised pruning-based DSE strategies, where evaluation of inefficient mappings is discarded and almost the same quality of mappings is produced. Next, we present an exhaustive and pruning-based DSE strategies.

Homogeneous Exhaustive Design Space Exploration (HomEDSE)

In HomEDSE, the exploration of mappings follows a set of steps described subsequently. An application with one actor (a_1) to be mapped on GPP tiles has only one unique actor to tile mapping, which is computed from equation Eq. 7.4. An application with two actors (a_1,a_2) has two unique mappings that is computed from equation Eq. 7.5. One mapping contains actors on separate tiles (${}^1C_0$ implies that from the remaining one actor a_1, it is not chosen to combine it with actor a_2) and another on the same tile (${}^1C_1$ implies that actor a_1 is chosen to combine it with actor a_2). Similarly, for an application with three actors (a_1,a_2,a_3), the unique mappings are computed from equation Eq. 7.6.

First, actor a_3 is mapped separately, i.e., not combined with others (from the remaining two actors a_1 & a_2, none is chosen to combine with a_3, indicated as $^2\mathrm{C}_0$) and remaining two actors are mapped by using equation Eq. 7.5 (f_{EDSE}(2,a_1,a_2)), providing two unique mappings. Then, from the remaining two actors one actor is chosen to combine with actor a_3 ($^2\mathrm{C}_1$) and the remaining actor is mapped separately, providing two unique mappings. Next, from the remaining two actors, both are chosen to combine with actor a_3, providing one unique mapping. Thus, for an application with three actors, a total of five unique actors to tiles mappings are evaluated. In the similar manner, for an application with four actors (a_1,a_2,a_3,a_4), all the unique mappings are computed from equation Eq. 7.7 and we get a total of 15 unique mappings.

The equations are extended in the similar manner to evaluate all the unique mappings when an application contains large number of actors. For an application with n actors (a_1,a_2,...,a_n), the mappings can be computed from equation Eq. 7.8. It can be observed that when computing mappings for larger number of actors, the mappings computed at lower number of actors are used, such as $f_{EDSE}(n-1,a_1,a_2,...,a_{n-1})$ in $f_{EDSE}(n,a_1,a_2,...,a_n)$.

$$f_{EDSE}(1, a_1) = 1 \tag{Eq. 7.4}$$

$$f_{EDSE}(2, a_1, a_2) = {}^1C_0 \times f_{EDSE}(1, a_1) + {}^1C_1 \tag{Eq. 7.5}$$

$$\begin{aligned} f_{EDSE}(3, a_1, a_2, a_3) &= {}^2C_0 \times f_{EDSE}(2, a_1, a_2) \\ &+ {}^2C_1 \times f_{EDSE}(1, remain\_actor) + {}^2C_2 \end{aligned} \tag{Eq. 7.6}$$

$$\begin{aligned} f_{EDSE}(4, a_1, a_2, a_3, a_4) &= {}^3C_0 \times f_{EDSE}(3, a_1, a_2, a_3) \\ &+ {}^3C_1 \times f_{EDSE}(2, remain\_actors) \\ &+ {}^3C_2 \times f_{EDSE}(1, remain\_actor) + {}^3C_3 \end{aligned} \tag{Eq. 7.7}$$

$$\vdots$$

$$\begin{aligned}
f_{EDSE}(n, a_1, a_2, ..., a_n) = {}^{(n-1)}C_0 \times f_{EDSE}(n-1, a_1, a_2, ..., a_{n-1}) \\
+{}^{(n-1)}C_1 \times f_{EDSE}(n-2, remain\_actors) \\
+{}^{(n-1)}C_2 \times f_{EDSE}(n-3, remain\_actors) \\
\vdots \\
+{}^{(n-1)}C_{n-2} \times f_{EDSE}(1, remain\_actor) + {}^{(n-1)}C_{n-1}
\end{aligned} \qquad \text{(Eq. 7.8)}$$

Homogeneous Pruning-based Design Space Exploration (HomPDSE)

In HomPDSE strategy, first 1_actor-to-1_GPP_tile mapping is evaluated, where n actors of the application are mapped onto n GPP tiles so that each GPP tile contains exactly one actor and the edges are mapped onto connections. The mapping is added to a mapping set M. Then, mappings at reduced tile count ($p = n-1$), i.e. mappings using $(n-1)$ GPP tiles are evaluated by Algorithm 8. The algorithm takes the best mapping using $(p+1)$ GPP tiles as input and evaluates mappings using p GPP tiles. First, $(p+1)$ GPP tiles containing actor(s) are selected. Then, for each pair of selected tiles, all the actors from one GPP tile are moved to another to generate a new mapping. Each generated mapping is added to a mapping set M after computing its throughput and energy consumption. For the selected $(p+1)$ GPP tiles, the algorithm finds $(p+1)$-choose-2 (${}^{(p+1)}\mathrm{C}_2$) unique pairs and thus evaluates the same number of mappings using p GPP tiles, where $0 < p < n$.

Out of all the evaluated mappings using p GPP tiles, the maximum throughput mapping is selected as the best mapping to evaluate mappings at further reduced tile

Algorithm 8: GPP Tiles Mappings Evaluation at Reduced Tile Count

```
Input: Best mapping $\alpha$ using $(p+1)$ GPP tiles.
Output: Mappings using $p$ GPP tiles.
Select $p+1$ GPP tiles ($\in T$) containing actor(s);
for each unique pair of selected tiles do
    Move actor(s) from one GPP tile to another to generate a new mapping $\beta$;
    Compute throughput and energyConsumption of $\beta$;
    Add $\beta$ with its throughput and energyConsumption to set $M$;
end
```

count, i.e. mappings using $(p-1)$ GPP tiles by following the steps of Algorithm 8. The same process is repeated until the tile count reduces to one. Thus, all the mappings using different number of GPP tiles get stored into the mapping set M. We have assumed that GPP implementation alternative is available for all the actors. However, this assumption can easily be removed by allocating the actors to their first available implementation alternatives.

7.2.1.2 Evaluating Heterogeneous Tiles Mappings

The heterogeneous tiles combination mappings are evaluated by using the GPP tiles mappings (M) obtained in the previous step. Such mappings are possible only when implementation alternatives other than GPP tiles are also available. To evaluate the combination mappings, we have devised two exploration strategies. In one, the exploration is performed exhaustively in order to evaluate all the possible mappings. The exhaustive exploration is not scalable as the number of mappings grows exponentially with the number of actors or tiles. In another strategy, we perform the exploration by pruning among the mappings. The evaluated mappings are added to the same mapping set M. Next, we introduce the two strategies.

Heterogeneous Exhaustive Design Space Exploration (HetEDSE)

In HetEDSE, the exploration is performed by using Algorithm 9. The algorithm evaluates all the possible combination mappings corresponding to each mapping in set M. This

Algorithm 9: Exhaustive Heterogeneous Tiles Combination Mappings Evaluation

```
Input: GPP tiles mappings $M$.
Output: Heterogeneous tiles combination mappings to be added to set $M$.
for each GPP tiles mapping $\alpha$ ($\in M$) do
    findHeterogTilesCombMappings($\alpha$, $t_1$);
end

function findHeterogTilesCombMappings(Mapping $\beta$, Tile startGPPTile)
if startGPPTile == lastGPPTile+1 then
    return;
end
for Tile i = firstGPPTile to lastGPPTile (in current mapping) do
    for each implementation alternative $\kappa$ other than GPP (e.g., DSP, ACC, RH tiles) do
        if tile i contains actor(s) $\in A$ and all have their implementation alternatives as $\kappa$ then
            Move actor(s) of tile $i$ to a $\kappa$-type tile having no previous actor to generate a new
            mapping $\alpha$;
            Compute throughput and energyConsumption of $\alpha$;
            Add $\alpha$ with its throughput and energyConsumption to set $M$;
            findHeterogTilesCombMappings($\alpha$, $i$+1);
        end
    end
end
end function
```

approach explores more number of mappings and decreases the chance of missing the maximum through mapping at different combinations, but the exploration overhead is high.

Heterogeneous Pruning-based Design Space Exploration (HetPDSE)

In HetPDSE, the exploration is performed by using Algorithm 10. At each tile count (*tileCount*), the maximum throughput mapping using GPP tiles is selected to generate mappings at different processing tile-combinations. For the selected mapping, the actors on each GPP tile are moved to another tile type (implementation alternative) in order to generate a new mapping provided all the actors on the GPP tile can be supported on the other tile type. The actors moving condition avoids the evaluation of mappings using non-supported tile-combinations. The generated mapping with its throughput and

Algorithm 10: Pruning-based Heterogeneous Tiles Combination Mappings Evaluation

```
Input: GPP Tiles Mappings $M$.
Output: Heterogeneous tiles combination mappings to be added to set $M$.
for tileCount = n (number of actors in the AG); tileCount >= 1; tileCount −− do
    Select maximum throughput mapping $\gamma$ using tileCount GPP tiles from set $M$;
    $maxNrTileTypesUsed = 1$ // only GPP tiles used;
    repeat
        Initialize the mapping set $S$, i.e., $S = \{\ \}$;
        for each GPP tile $t$ ($\in T$) in the current mapping $\gamma$ do
            for each implementation alternative $\kappa$ other than GPP (e.g., DSP, ACC, RH
            tiles) do
                if $t$ contains actor(s) $\in A$ and all have their implementation alternatives as $\kappa$
                then
                    Move actor(s) of tile $i$ to a $\kappa$-type tile having no previous actor to generate
                    a new mapping $\delta$;
                    Compute throughput and energyConsumption of $\delta$;
                    Add $\delta$ with its throughput and energyConsumption to set $S$ and to global
                    set $M$;
                    Move actor(s) back on the initial tile $t$ to reset $\gamma$;
                end
            end
        end
        Select maximum throughput mapping from set $S$ and assign as current mapping $\gamma$;
        $maxNrTileTypesUsed++$;
    until $maxNrTileTypesUsed \leq tileCount$;
end
```

energy consumption is added to set M and temporary set S. The mappings at next possible tile-combinations are evaluated by selecting the maximum throughput mapping from the temporary set S. By selecting the maximum throughput mapping at different places in the algorithm, evaluation of inefficient mappings is discarded.

7.2.1.3 Selecting and Storing Best Mapping at Each Processing Resource Combination

At each possible processing tile-combination, we get a number of mappings. This step selects the maximum throughput mapping and minimum energy consumption mapping at each tile-combination, and stores them into the *mappings with throughput & energy* database (*MTED*) (Fig. 7.5). In cases when both the maximum throughput and minimum energy consumption mapping are the same, only one mapping is stored. The stored mappings are sorted by number of tiles used by them in increasing order. The number of used tiles are referred to as tile count. At each tile count, the mappings get stored in increasing order of heterogeneity as explained in Algorithm 10. Increasing heterogeneity implies use of more number of tile types.

Storing the mappings in such order facilitates for run-time selection from lower tile count to higher tile count and in increasing order of heterogeneity at each tile count. The run-time approach finds a throughput-satisfying mapping using the minimum number of tiles (tile count) and having minimum energy consumption. While evaluating mappings by Algorithm 10, it might be possible that all the possible tile-combinations are not covered because of the pruning consideration to speed up the exploration. In such cases, at run-time we need to look for a combination that is subset of the covered combination. The run-time algorithm is described later in Section 7.2.2.

7.2.1.4 Optimization

Amongst the stored mappings in the database *MTED*, it might be possible that some of them are sub-optimal. The sub-optimal mappings require more number of processing processing tiles as compared to others and have less throughput (performance) and higher energy consumption. For example, for an application, a mapping requiring 3 GPP & 1 ACC tiles might have less throughput and high energy consumption as compared to a mapping requiring only 2 GPP & 1 ACC tiles. The former mapping is sub-optimal and it has to cater for larger communication overhead without much gain in parallel processing and thus provides less throughput and consumes high energy. There is no point in keeping such sub-optimal mappings. So, we perform an *optimization* on *MTED* to discard all such mappings in order to store only Pareto-optimal mappings as *Optimal Mappings with Throughput & Energy (OMTED)* (Fig. 7.4).

The concepts of Pareto algebra has been used to find the Pareto-optimal mappings [163]. In *optimization*, we compare throughput and energy consumption of mappings requiring higher number of tiles to ones requiring lower number of tiles. If throughput of a mapping using higher number of tiles is the same or smaller than the throughput of a mapping using lower number of tiles, energy consumption in latter mapping is the same or lower than the former mapping and processing tiles in the latter mapping are a subset of processing tiles in the former mapping, then the former mapping is discarded. The same process is performed for each processing tile-combination to discard all the sub-optimal mappings. The optimization result includes Pareto-optimal mappings and each such mapping is better than another in terms of throughput, energy consumption or resource usage. Keeping only the optimal mappings reduces memory requirement to store them and overhead in selecting the best mapping since the run-time mapping strategy needs to select from a relatively smaller set of mappings.

Design-time Analysis: Complexity

The design-time analysis complexity in terms of number of actors n, number of implementation alternatives μ and max_hop_distance h has been computed. The worst-case complexity (C) is determined by the total number of evaluated mappings (M) in the DSE flow (Figure 7.5) when all the actors have μ implementation alternatives. We have computed the total mappings when pruning-based DSE is performed during both the homogeneous and heterogeneous tiles mappings evaluation. For a given value of n, μ and h, the total number of mappings evaluated over the DSE loops is calculated by Equation Eq. 7.9. The number of homogeneous and heterogeneous tiles mappings is evaluated by Equation Eq. 7.10 and Eq. 7.11, respectively.

$$C = h \times [nrHomogeneousTilesMappings + nrHeterogeneousTilesMappings] \tag{Eq. 7.9}$$

$$nrHomogeneousTilesMappings = 1 + \sum_{p=1}^{n-1} ({}^{(p+1)}C_2) = 1 + \sum_{p=1}^{n-1} \left(\frac{p^2}{2} + \frac{p}{2} \right) = 1 + \frac{n^3 - n}{6} \qquad \text{(Eq. 7.10)}$$

$$\begin{aligned} nrHeterogeneousTilesMappings &= (\mu - 1) \sum_{p=1}^{n} \{p + (p-1) + (p-2) + ... + 2 + 1\} \\ &= (\mu - 1) \sum_{p=1}^{n} \left(\frac{p^2}{2} + \frac{p}{2} \right) = (\mu - 1) \left(\frac{n^3}{6} + \frac{n^2}{2} + \frac{2n}{6} \right) \end{aligned} \qquad \text{(Eq. 7.11)}$$

Thus, the total number of mappings can be calculated as follows.

$$C = h \times \left[1 + \frac{n^3 - n}{6} + (\mu - 1) \left(\frac{n^3}{6} + \frac{n^2}{2} + \frac{2n}{6} \right) \right] = h \times \left[\frac{\mu n^3}{6} + \frac{(\mu - 1)n^2}{2} + \frac{(2\mu - 3)n}{6} + 1 \right] \qquad \text{(Eq. 7.12)}$$

In Equation Eq. 7.10, ${}^{(p+1)}\mathrm{C}_2$ is the number of unique pair of GPP tiles at tile count of $p + 1$. Each pair forms a mapping using p GPP tiles. Heterogeneous tiles mappings are possible only when any actor has more than one implementation alternative, i.e. $\mu > 1$. So, Equation Eq. 7.11 is valid for $\mu > 1$. Total number of mappings can be calculated from Equation Eq. 7.12, which has complexity of $\mathrm{O}(h\mu n^3)$. The existing and exhaustive exploration strategies evaluate more number of mappings as compared to the pruning-based strategy and thus have complexity of higher orders. The mappings evaluated by existing strategies are discussed in Section 7.4 and compared with our strategies.

Design-time Analysis for a Given Platform Size

The DSE strategy presented in Figure 7.5 considers a generic MPSoC platform. However, for a given platform (PG) containing smaller number of tiles than the number of actors in the application (AG), the mappings with more number of tiles than present in the given platform will never be used. Such mappings have been referred to as infeasible mappings for the given platform. The DSE process can be speeded up by discarding the evaluation of such infeasible mappings.

Evaluation of infeasible mappings is avoided by extending the DSE flow presented in Figure 7.5. The number of tiles (nrTiles) in the given platform is taken as one additional input to the DSE flow. Homogeneous tiles mappings are evaluated in the similar manner. While evaluating heterogeneous tiles combination mappings, Algorithm 10 is modified to start the mappings evaluation starting from tile count value of *nrTiles* in order to discard evaluation of infeasible mappings. All the mappings using a maximum of *nrTiles* tiles are then selected and stored, which will be applicable to the given platform.

7.2.2 Run-time Mapping

The *Design-time Analysis* step performs all the compute intensive analysis and thus leaves for minimum computation at run-time. The generated mappings in the analysis are applicable to any target MPSoC platform considered at run-time as long as *i)* the target platform tile types are subset of the analyzed tile types and *ii)* the maximum distance between the chosen target platform tiles for mapping is less than or equal to the maximum separation for which the DSE was performed. Thus, no additional design-time analysis is needed in case of such different target platforms. This approach is analogues to *analyze once & run everywhere*, which is similar to Java's *write-once-run-everywhere* capability. Figure 7.6 shows a demonstration for three types of tiles (GPP, RH, ACC) analyzed during DSE with maximum separation between the tiles as 3 hop (hop_distance). The analyzed results will be applicable to the three shown target platforms as their tile types and max hop_distance are subset of the tile types and maximum separation considered during DSE.

Run-time mapping of throughput-constrained multimedia applications onto a platform is handled by the *Run-time Manager* (Figure 7.4). Out of many available processors in the platform, one of them is used as manager processor that is responsible for actor mapping, actor scheduling, platform resource control and configuration control. The resources status is updated at run-time when an actor is loaded in the platform in order

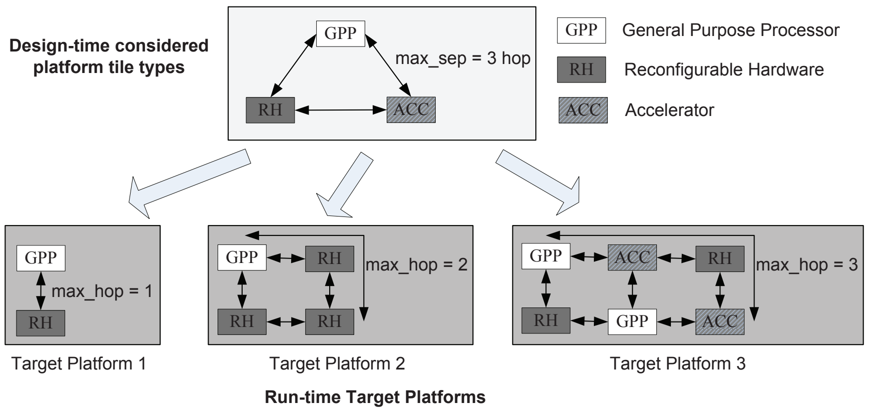


Figure 7.6: Analyze once & run everywhere demonstration.

to provide the manager processor with accurate knowledge of resource occupancy which is required for taking the mapping decision based on available resources at run-time. Run-time manager (RTM) maps the applications on the platform one after another, i.e. after accomplishing mapping for one application, it goes on to map the next application till all the applications are mapped. The strategy adopted by the RTM to map an application is presented in Algorithm 11. The strategy takes the application, its desired throughput, platform with updated resources' status and the optimized mapping storage $OMTED$ as input and selects the best mapping from the $OMTED$ depending upon the desired throughput and available platform tiles. The selected best mapping satisfies the throughput requirement, uses minimum resources and has minimum energy consumption. The platform is then configured based on the actors to tiles allocations provided in the selected mapping.

The RTM first finds the maximum number of tiles that might get used (Max_Tiles_Used) by the application and then the number of available tiles ($Tiles\_Available$) in the platform. A *mapping* satisfying the throughput constraint of the application ($\tau \leq thrMapping$) and having minimum $energyconsumption$ is selected from the $OMTED$ by iterating from tile count one to Max_Tiles_Iter. Maximum tiles iteration value Max_Tiles_Iter is calculated as minimum of $Tiles\_Available$ & Max_Tiles_Used in order to restrict unnec-

Algorithm 11: Hybrid Run-time Mapping

```
Input: Application $AG$, Required throughput $\tau$, Platform $PG$, Optimized mapping database
       $OMTED$.
Output: The best mapping satisfying the throughput-constraint $\tau$.
$Max\_Tiles\_Used$ = nrActors($AG$); $Tiles\_Available$ = nrAvailTiles($PG$); $Max\_Tiles\_Iter$ =
0; $tile\_count = 1$;
if $Tiles\_Available > 0$ then
    $Max\_Tiles\_Iter$ = min($Max\_Tiles\_Used$, $Tiles\_Available$);
    repeat
        for each mapping $\phi$ using tile_count tiles in OMTED do
            Select closest available $tile\_count$ tiles used by $\phi$ in $PG$;
            $hop\_max$ = findMaximumHop(selected tiles);
            $thrMapping$ = Find($OMTED$, $AG$, $tile\_count$, $\phi$, $hop\_max$);
            if $\tau \leq thrMapping$ then
                $Mapping\_list$ = Find all throughput satisfying mappings using the same
                resource combination as of $\phi$ from $OMTED$;
                Select the mapping having minimum energyconsumption from $Mapping\_list$
                and exit;
            end
        end
        $tile\_count$++;
    until $tile\_count \leq Max\_Tiles\_Iter$;
    No mapping found;
else
    Application can't be supported, i.e., no mapping found;
end
```

essary search in the *OMTED*. For each mapping using *tile_count* tiles, first, the RTM selects closest available tiles in the platform, then finds maximum hop_distance (*hop_max*) between the selected tiles, and finally, throughput of the mapping (*thrMapping*) to be checked against the throughput constraint τ. As soon as a throughput satisfying mapping ϕ is found ($\tau \leq thrMapping$), all the throughput satisfying mappings using the same resource combination as of ϕ are found and added into a mapping list (*Mapping_list*). Thereafter, the mapping having minimum energy consumption is selected from the mapping list and the platform is configured based on the selected mapping. If a throughput satisfying mapping is not found then the application cannot be supported on the platform with the available resources. In such case, the application mapping may be tried with relaxed throughput requirement in order to support it on the platform.

Throughput computation for a mapping takes much more time than the time to find the mapping, i.e. tasks to tiles allocations. Our RTM just selects the best mapping without involving throughput computation at run-time and thus accelerates the run-time mapping process. Further, the selected throughput satisfying mapping uses minimum number of tiles as search is performed from lower tile count to higher tile count. The selected mapping has minimum energy consumption as well. Therefore, the RTM performs effective and efficient mapping.

7.3 Implementing Hybrid Mapping

The hybrid mapping flow has been applied onto some applications to demonstrate how the flow first performs design-time analysis, and then maps the required applications onto a platform at run-time.

Design-time Analysis

The DSE step of design-time analysis evaluates multiple mappings for an application. Let us consider an application modeled with 3 actors ($a1$, $a2$ and $a3$) having implementation alternatives as GPP, DSP and ACC tiles for each of them. The DSE flow first considers a platform containing 3 tiles of each implementation alternative, and then evaluates mappings using GPP tiles followed by mappings using combinations of GPP, DSP and ACC tiles. The GPP, DSP and ACC tiles are represented in different shades as shown in Figure 7.7.

Mappings using only GPP tiles are evaluated by the HomPDSE strategy described in Section 7.2.1.1. First, 1_actor-to-1_GPP_tile mapping is evaluated where each GPP tile contains exactly one actor as shown in Figure 7.7 (top-left mapping). Only the used tiles of the mapping are shown. The edges are mapped on connections between the tiles

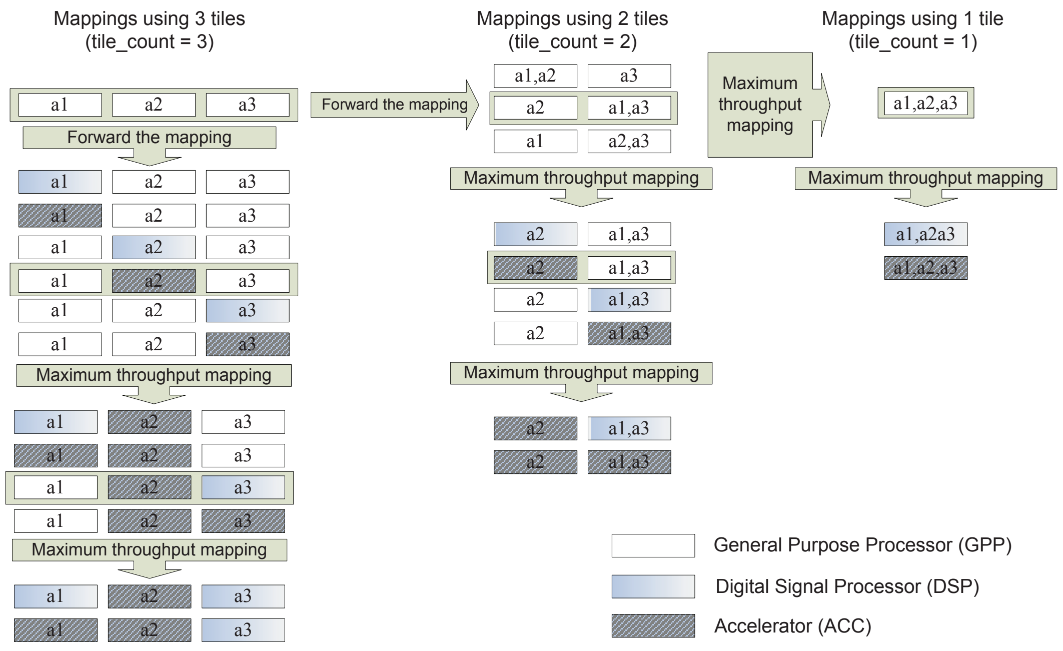

Mappings using 3 tiles (tile_count = 3)
Mappings using 2 tiles (tile_count = 2)
Mappings using 1 tile (tile_count = 1)
Forward the mapping
Maximum throughput mapping
a1
a2
a3
a1,a2
a1,a3
a2,a3
a1,a2,a3
General Purpose Processor (GPP)
Digital Signal Processor (DSP)
Accelerator (ACC)

Figure 7.7: Design space exploration for an application modeled with 3 actors a1, a2 and a3.

which we have not shown as we want to focus only on the number of mappings that depends upon placement of the actors. Here, for each mapping, the tiles are shown as linearly arranged as we just want to illustrate the DSE flow, whereas in the actual flow the separation between the tiles can be any fixed value of hop_distance. Next, mappings at one reduced tile count, i.e., mappings using 2 GPP tiles are evaluated by Algorithm 8. The algorithm finds 3 ($^3C_2$) unique pair of tiles containing actor(s) from 1_actor-to-1_GPP_tile mapping as shown in Figure 7.7. The maximum throughput mapping at each tile count is selected and forwarded to evaluate mappings at reduced tile count. We have considered the highlighted mapping as the maximum throughput one so the same is forwarded. We get a mapping using one GPP tile. Thus, the flow evaluates a total of 5 mappings, which is the same as the ones calculated from equation Eq. 7.10, i.e., [$1 + (3^3 - 3)/6$].

Mappings using combination of GPP, DSP and ACC tiles are evaluated by the Het-PDSE strategy (Algorithm 10) described in Section 7.2.1.2. At each tile count, Algorithm

10 takes maximum throughput mapping using GPP tiles as input and evaluates mappings using combination of tiles as shown in Figure 7.7. Each task is moved from GPP tile to DSP and ACC tiles to generate mappings. The maximum throughput mapping (highlighted one) is selected and forwarded to evaluate mappings at further tile combinations by moving only the tasks of GPP tiles to DSP and ACC tiles. The same process is repeated until all the tasks of GPP tiles are moved to DSP or ACC tiles. The algorithm evaluates a total of 20 mappings, which is the same as the ones calculated from equation Eq. 7.11 by putting n and μ equal to 3.

Similarly, DSE can be demonstrated for multimedia applications modeled with different number of actors. Similar demonstration can be performed by using other DSE strategies for evaluating homogeneous and heterogeneous tiles mappings. Let us assume that the target platform on which the applications need to be mapped is a 4×4 grid of tiles as shown in Figure 7.8. For this platform, the value of max_hop_distance is 6, so the DSE is repeated 6 times by considering platform tiles separated by hop_distance of 1 to 6.

Run-time Mapping

The run-time mapping of the analyzed applications on the target platform is handled by the run-time strategy presented in Algorithm 11. The applications are mapped one after another. For each application, the strategy selects the best mapping from the OMTED subject to desired throughput and available platform tiles. Let some applications be already mapped on the platform and they are using the busy tiles as shown in Figure 7.8. Run-time mapping of H.263 decoder (Figure 7.2) and DSE demonstrated application (modeled with 3 actors a1, a2 and a3) on the available tiles is shown in Figure 7.8. Let us assume that for H.263 decoder and the demonstrated application, throughput satisfying mappings using 3 and 2 tiles respectively are found which uses different tile type combinations.

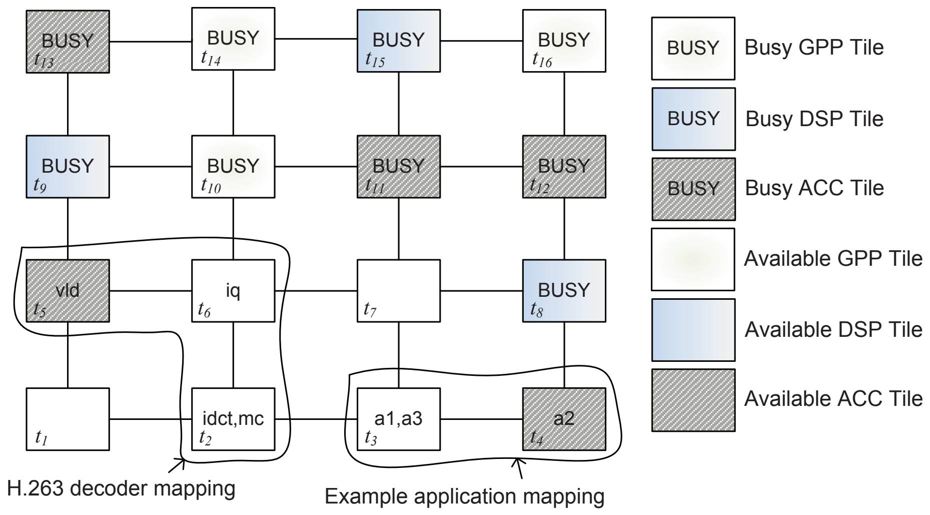


Figure 7.8: Run-time mapping of H.263 decoder (4 actors) and the example application (3 actors a1, a2 and a3).

The four actors *vld, iq, idct* & *mc* of H.263 decoder are mapped onto the 3 closest available tiles t_2, t_5 & t_6 based on the allocations provided in its found mapping as shown in Figure 7.8. In the found mapping, all the edges are separated by a hop_distance of 2. So, mapping the actors on the available tiles as shown will satisfy the throughput constraint for sure as some edges will be mapped at lower hop_distances (lower latencies). Edges are mapped on the connections between the tiles. Similarly, three actors a1, a2 & a3 of the demonstrated application are mapped onto 2 closest available tiles t_3 & t_4 based on its found mapping, as shown in the Figure 7.8.

7.4 Performance Evaluation

The proposed hybrid mapping strategy has been implemented as an extension of the publicly available SDF^3 tool set [93]. As a benchmark to evaluate the run-time and quality of the strategy, models of real-life multimedia applications H.263 decoder (4 actors), H.263 encoder (5 actors), JPEG decoder (6 actors), MP3 decoder (14 actors) and models of synthetic applications containing varying number of actors have been considered. Experiments are performed on a Core 2 Duo processor at 3.16 GHz.

The same generic platform graph is considered to evaluate the different strategies for an application. In the platform, the number of tiles and their types depend upon the number of actors and their implementation alternatives provided in the application. We consider tile-based architecture but any other type of architecture can also be considered based on the known latencies between the tiles as discussed earlier.

The same generic platform graph is considered to evaluate the different strategies for an application. In the platform, the number of tiles and their types depend upon the number of actors and their implementation alternatives provided in the application. We consider tile-based architecture but any other type of architecture can also be considered based on the known latencies between the tiles as discussed earlier. For an actor, the implementation alternative could be GPP, accelerator, RH etc. ARM7TDMI [164] is used as GPP. The accelerator for each actor is different as it is customized for a specific task to be performed by the actor. The RH can be configured to support actors according to their requirement and as an accelerator. The considered applications contain some common actors and we have considered the same RH for an actor. The common actors *video length decoding* (*vld*) [165], *inverse quantization* (*iq*) [166], *inverse discrete cosine transform* (*idct*) [167], *motion compensation* (*mc*), *motion estimation* (*me*) and *Deblocking* [168] are considered to have their one of the implementation alternatives as RH. While performing simulation, execution time and power consumption of different actors are considered based on their mapping on different types of tiles.

In particular, while evaluating homogeneous tiles mappings, we present results obtained from our design-time pruning-based (HomPDSE) exploration flow and compare them to that of the exhaustive exploration (HomEDSE) flow, the flow presented in [138] and [141]. While evaluating heterogeneous tiles mappings, we present results obtained from HetEDSE and HetPDSE flows and compare them to the existing flows. For evaluating the homogeneous and heterogeneous tiles mappings when HomPDSE and HetPDSE are employed respectively, the overall approach is referred to as heuristic DSE (HDSE) and when HomEDSE and HetEDSE are employed, the approach is referred to as exhaustive DSE (EDSE). We implemented the flow in [138], [141], EDSE and HDSE flows with similar steps in order to make a fair comparison amongst them. The flow in [141] is applied to scenarios, where each scenario contains a different version of the same application. The different versions model different behavior of an application at different times. We consider a single scenario, i.e., a single version of the application that has always the same behavior. So, mappings obtained with this flow can be fairly compared with our DSE flows. We have compared HDSE flow with above mentioned flows as they also perform exploration to evaluate mappings providing different throughput values and we target throughput aware run-time mapping. The results from the hybrid run-time mapping (HRM) technique are compared to that of run-time techniques presented in [129] and [J-1].

7.4.1 Design Space Exploration

Table 7.3 shows the design-time HDSE results for the H.263 decoder (4 actors) at max_hop_distance of 4 when each actor has two implementation alternatives ARM and RH. The DSE flow runs 4 times from hop_distance of 1 (hop_1) to 4 (hop_4). For each run, the numbers of evaluated and best mappings at different tile-combinations are shown as *nrMaps* and *nrBestMaps* respectively. A total of 31 mappings are evaluated, which is the same as calculated from equation Eq. 7.12 with n and μ values as 4 and 2 respectively. At each tile-combinations, the best mappings (*nrBestMaps*) are chosen from the evaluated mappings (*nrMaps*). The best mappings excel in throughput or energy consumption. We can get more than one best mapping as the evaluated mappings might have throughput-energy trade-offs. At each hop, the best mappings' throughput & energy consumption is shown. The last two rows (for tile count value of 1) are empty for hop_1 through hop_4

Tile	Tile			Best mappings' throughput ($\times 10^{-10}$/time-units) & energy consumption ($\times 10^{-3} mJ$)				
Count	Combinations	nrMaps	nrBestMaps	hop_0	hop_1	hop_2	hop_3	hop_4
4	4ARM	1	1	×	28,655 & 393	28,616 & 515	28,578 & 638	28,540 & 761
	3ARM & 1RH	4	1	×	29,170 & 301	29,130 & 424	29,090 & 547	29,050 & 670
	2ARM & 2RH	3	2	×	29,170 & 249	29,130 & 372	29,090 & 495	29,051 & 618
				×	29,612 & 268	29,571 & 390	29,530 & 513	29,489 & 636
	1ARM & 3RH	2	1	×	29,612 & 216	29,571 & 369	29,530 & 461	29,489 & 584
	4RH	1	1	×	29,612 & 194	29,571 & 317	29,530 & 439	29,489 & 562
3	3ARM	6	1	×	62,669 & 352	62,665 & 434	62,661 & 515	62,657 & 597
	2ARM & 1RH	3	1	×	67,387 & 227	67,382 & 309	67,378 & 390	67,373 & 472
	1ARM & 2RH	2	2	×	67,387 & 175	67,383 & 257	67,378 & 338	67,374 & 420
				×	69,970 & 205	69,965 & 287	69,960 & 369	69,956 & 450
	3RH	1	1	×	69,970 & 153	69,965 & 235	69,960 & 317	69,956 & 398
2	2ARM	3	1	×	**91,585** & **311**	**91,583** & **352**	**91,580** & **393**	**91,578** & **434**
	1ARM & 1RH	2	1	×	**123,230** & **164**	**123,228** & **205**	**123,227** & **246**	**123,226** & **287**
	2RH	1	1	×	**123,230** & **112**	**123,228** & **153**	**123,227** & **194**	**123,226** & **235**
1	1ARM	1	1	**73,961** & **270**	×	×	×	×
	1RH	1	1	**106,204** & **71**	×	×	×	×

Table 7.3: DSE results for H.263 decoder.

as hop_distance between mapped actors on the single tile will be zero. This hop_distance is referred to as hop_0 and it is not applicable when actors will mapped on more than one tile, denoted as ×. Similar DSE results have been obtained for other multimedia applications. The results can easily be extended for higher hops by taking a large value of max_hop_distance, which can cater for larger future target platforms. At run-time, one can select a mapping having maximum throughput and minimum energy consumption depending upon the available tiles and maximum hop_distance between them.

The Pareto-optimal mappings are highlighted in Table 7.3. These mappings require less number of tiles and provide the same or better performance (throughput and energy consumption). It can be seen in Table 7.3 that the mappings using 3 or 4 tiles have worst performance than mappings using only 2 tiles. This is because of larger communication overhead while using more number of tiles and not gaining much in parallel processing. Similar optimization results have been obtained for other multimedia applications.

We have applied the exhaustive DSE (EDSE), the flow in [141] and the HDSE flow to find the number of mapping to be evaluated by them. The EDSE flow evaluates all the possible mappings at different tile-combinations. The number of mappings evaluated by EDSE flow increases exponentially with the number of actors. Further, the number of mappings increases even more when the implementation alternatives of actors, i.e. number of tile types on which the actors can be supported get increased. For n actors having $nrTileTypes$ implementation alternatives for each of them, the EDSE flow considers a platform containing n tiles of each implementation alternative and uses a maximum of n tiles in mappings to be evaluated. The total number of mappings are calculated as the number of ways placing n labeled balls into n unlabeled (but $nrTileTypes$-colored) boxes, where balls and boxes represent tasks and tiles respectively. The number of mappings by the DSE flow in [141] are limited by X times the number of actors times the number of tiles, where X is the maximum number of partial bindings that is carried over to the next iteration for evaluating the mappings. The HDSE flow considers pruning where maximum throughput mapping is selected for further evaluation and thus limits the number of mappings to be evaluated.

Table 7.4 shows the number of mappings evaluated by the EDSE, [141] and HDSE flow as the number of actors ($nrActors$) increases at different number of available implementation alternatives ($nrTileTypes$) for each of the actor. At $nrTileTypes$ equal to 1, the number of mappings evaluated by the EDSE at increasing values of $nrActors$ follows bell numbers which represents the number of ways of placing $nrActors$ labeled balls into $nrActors$ indistinguishable boxes [169]. The number of mappings by [141] flow is shown for X equal to 10. The number of mappings increases with the value of X and it may lead to an explosion in the number of mappings. The number of mappings evaluated by HDSE follows Equation Eq. 7.12. For $nrActors$ and $nrTileTypes$ equal to 14 and 3 respectively, the EDSE evaluates 461,101,962,108 mappings. If we assume 1 millisecond (ms) to evaluate one mapping then it is going to take around 15 years in evaluating all the mappings. In our experiments, evaluation of each mapping takes close to 250 ms. Thus, EDSE is not scalable and evaluation is not always feasible.

	EDSE Flow			Ref. [141] Flow			HDSE Flow		
	nrTileTypes			nrTileTypes			nrTileTypes		
nrActors	1	2	3	1	2	3	1	2	3
1	1	2	3	1	2	3	1	2	3
2	2	6	12	6	20	42	2	6	10
3	5	22	57	39	102	180	5	15	25
4	15	94	309	100	132	372	11	31	51
5	52	454	1,866	180	410	615	21	56	91
6	203	2,430	12,351	282	612	918	36	92	148
7	877	14,214	88,563	406	854	1281	57	141	225
8	4,140	89,918	681,870	552	1136	1704	85	205	325
9	21,147	610,182	5,597,643	720	1458	2187	121	286	451
10	115,975	4,412,798	48,718,569	910	1820	2730	166	386	606
14	190,899,322	20,732,504,062	461,101,962,108	1834	3668	5502	456	1,016	1,576

Table 7.4: Number of mappings by Exhaustive DSE (EDSE), Ref. [141] DSE and Heuristic DSE (HDSE) at different number of actors (nrActors) and their available implementation alternatives (nrTileTypes).

7.4.2 Speed Up and Quality of Results

The HDSE has been employed to speed up the exploration process while providing almost the same quality of mappings as of the EDSE. The exploration flows HDSE and EDSE are applied to 100 random applications modeled as SDFGs with 4, 5, 6 and 7 actors having their implementation alternatives as ARM and RH tiles generated randomly. For each application, the best mapping at each resource combination has been captured. Figure 7.9 shows the quality (throughput) of the best mapping at 2 ARM and 1 RH tiles resource combination for all the 100 applications when tiles are assumed to be separated by a fixed hop_distance. The best mapping throughput obtained by HDSE is normalized with respect to (w.r.t.) EDSE. The normalized throughput values are plotted after sorting them in descending order.

It has been observed that loss in quality of mappings is more when the number of actors increases. Further, the HDSE provides the same best mappings as of EDSE for more than 80% of the applications. Similar behavior is obtained at other resource combinations. Thus, we can say that for most of the applications, we get the same quality

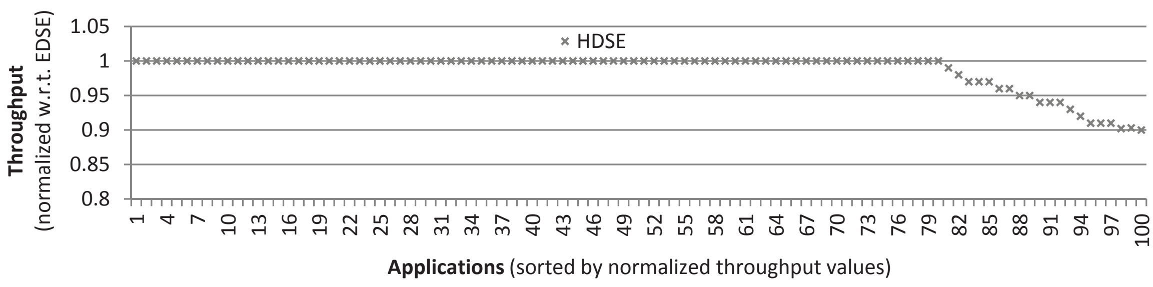


Figure 7.9: Quality of mappings by HDSE over EDSE for 100 random applications.

of mappings. It can be observed that for applications where we don't get the same quality of mappings by HDSE and EDSE, the quality varies only by 10%. So, in case of 10% relaxed throughput constraint at run-time, the mappings generated by HDSE will be acceptable for all the applications. The HDSE flow decreases the chances of missing the best mappings at different tiles combinations as it starts from a maximum throughput mapping to find mappings at each possible tiles combination.

Figure 7.10 shows the speed up obtained by HDSE over the EDSE for the same set of applications. The applications are sorted by the number of actors within them. It can be observed that HDSE is faster over the EDSE for all the applications. It also can be observed that as the number of actors in the applications increases, the speed up obtained by HDSE increases as the strategy evaluates lesser number of mappings by adopting pruning for evaluating both the homogeneous and heterogeneous tiles mappings. On an average, the HDSE is faster by 18× when compared to EDSE. For large size applications, exhaustive exploration time may be days or weeks so HDSE needs to be employed for performing exploration within a reasonable time.

We have applied EDSE and HDSE flows on multimedia applications H.263 decoder (4 actors), H.263 encoder (5 actors) and JPEG decoder (6 actors) too. It has been observed that for H.263 decoder/encoder the best mapping at different tiles combinations is the same by all the flows, whereas for JPEG decoder, HDSE misses best mapping at a few tiles combinations.

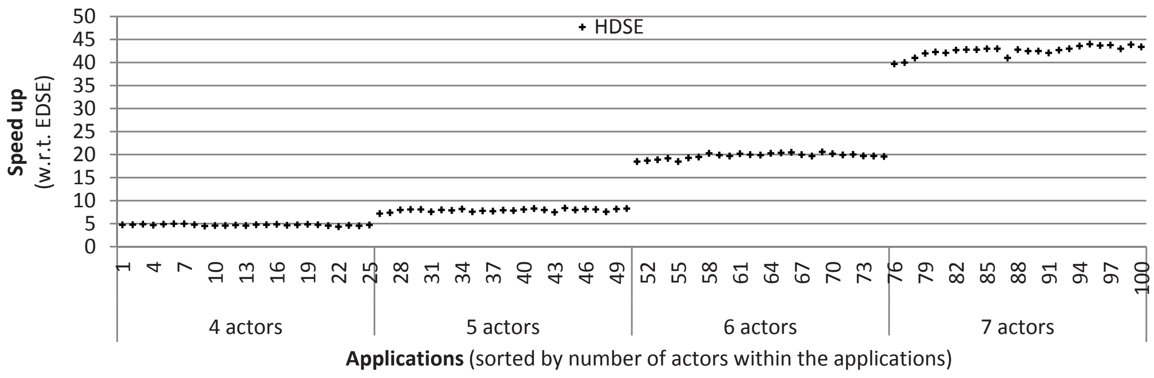


Figure 7.10: Speed up obtained by HDSE over EDSE for 100 random applications.

7.4.3 Design Space Exploration for a Given Platform

We performed multimedia applications DSE for given platforms that may contain any arbitrary number of tiles. Table 7.5 shows the DSE results for multimedia applications H.263 decoder, H.263 encoder and JPEG decoder for platforms containing 1×2, 2×2, 3×3 and 4×4 grid of tiles. For each platform, exploration time (milliseconds) and the best mapping throughput ($\times 10^{-12}$/time-units) has been tabulated when the exploration approach of [141], EDSE and HDSE have been employed. The different platform tiles are ARM tiles, i.e homogeneous platforms are considered. So, the EDSE and HDSE explore only homogeneous tiles mappings by employing HomEDSE and HomPDSE, respectively.

The number of evaluated mappings by the approach of [141] depends upon the number of tiles present in the platform. So, for larger platforms (containing more number of tiles), the approach of [141] evaluates higher number of mappings and thus shows increased exploration time as shown in the Table 7.5. The approach evaluates some duplicate mappings which differ in only placement of actors on different tiles providing the same throughput. So, in some cases, it evaluates more number of mappings (including duplicates) than the EDSE and thus takes more time than the EDSE as shown in Table 7.5.

The EDSE flow evaluates all the possible mappings without any duplicate ones and

		Exploration Time (ms)			Best Mapping Throughput	
Application	Platform	Ref. [141]	EDSE	HDSE	Ref. [141]	EDSE & HDSE
H.263 decoder (4 actors)	1×2	7,243	4,739	2,894	7,352,890	9,138,520
	2×2	11,479	9,476	5,787	7,396,120	9,158,520
	3×3	19,010	18,956	11,572	7,396,120	9,158,520
	4×4	24,738	28,444	17,358	7,396,120	9,158,520
H.263 encoder (5 actors)	1×2	10,576	18,559	5,817	649,689	794,199
	2×2	21,918	37,128	11,633	662,473	941,289
	3×3	38,659	74,154	23,265	662,473	941,289
	4×4	45,378	111,367	34,897	662,473	941,289
JPEG decoder (6 actors)	1×2	13,443	11,497	1,690	363,423,000	678,426,000
	2×2	23,175	22,847	3,345	384,225,000	678,426,000
	3×3	42,341	45,873	6,765	384,225,000	678,426,000
	4×4	56,983	68,783	10,152	384,225,000	678,426,000

Table 7.5: Multimedia applications DSE at different platforms for exploration time (ms) and best mapping throughput ($\times\ 10^{-12}$/time-units)

the HDSE flow performs pruning to discard evaluation of inefficient mappings. The EDSE and HDSE flows are executed in the similar manner. Larger platforms are covered by executing the flow repeatedly by considering higher separation (hop_distance) between the tiles. For 1×2, 2×2, 3×3 and 4×4 platforms, maximum hop_distance between the tiles is 1, 2, 4 and 6 respectively, so the flow is repeated maximum hop_distance times by increasing the delay of connections according to the hop_distance. In each execution of the flow, for H.263 decoder (4 actors), H.263 encoder (5 actors) and JPEG decoder (6 actors), the EDSE flow evaluates a total of 15, 52 and 203 mappings, whereas the HDSE flow evaluates a total of 11, 21 and 36 mappings respectively, as discussed earlier in Table 7.4. Table 7.5 shows the exploration time for complete execution of the EDSE and HDSE flow. It can be seen that HDSE shows lower exploration time than other DSE flows.

It can be observed from Table 7.5 that the HDSE flow does not miss the best throughput mapping despite requiring much lower time for exploration. On an average, for H.263 decoder, H.263 encoder and JPEG decoder, exploration time of the HDSE flow is lowered by 39%, 35% and 83% as compared to the flow in [141], and by 38%, 68% and 85% as

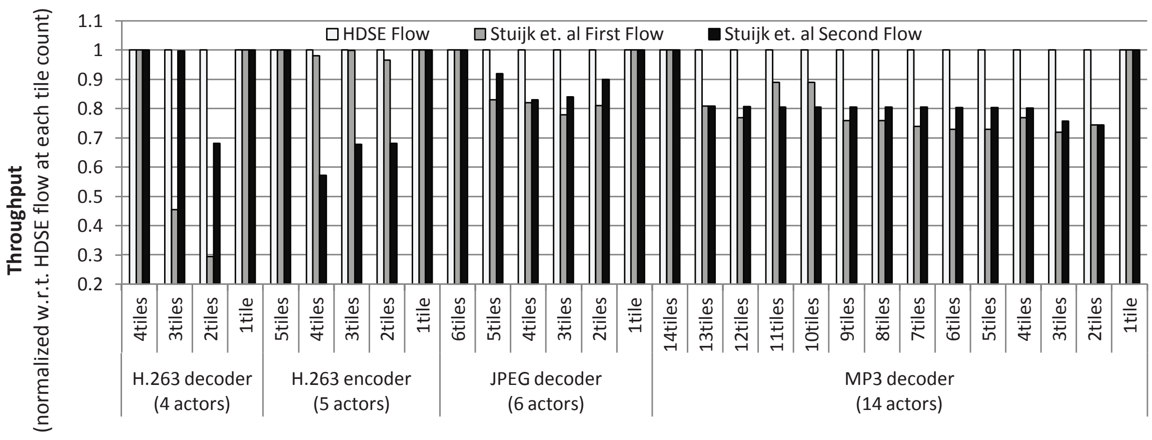


Figure 7.11: The best mapping throughput comparison at different platforms for different applications when HDSE Flow, Stuijk et. al First [138] and Second [141] Flows are employed.

compared to EDSE flow respectively. The difference in the number of explored mappings from EDSE and HDSE flow increases with the number of actors in the application and thus the difference in the exploration time as shown in Table 7.5. The exploration by EDSE is not feasible within a limit time when the number of actors are large as explained earlier. However, HDSE performs exploration within a limited time. The best mapping throughput for H.263 decoder, H.263 encoder and JPEG decoder is improved by 23%, 37% and 44% respectively over the approach of [141].

Figure 7.11 shows the throughput for the best mappings for multimedia applications where different number of ARM tiles is used for HDSE flow, the flow presented in [138] and [141]. The throughput at each tile count (number of used tiles by the mappings) has been normalized with respect to (w.r.t.) the HDSE flow. It can be observed that the HDSE flow always provides better quality (throughput) of mappings at all the tile counts. For each application, the same best mapping is obtained by all the flows at platforms containing one (1tile) and the same number of tiles as the number of actors.

We also performed DSE of multimedia applications for given platforms containing different types of tiles such as GPP, DSP and RH tiles. For a given platform containing

5 GPP and 5 RH tiles, the HDSE flow explores a total of 56 mappings for H.263 encoder (5 actors) when each actor has its implementation alternatives as GPP and RH tiles. The exploration took a run-time of 35,178 ms. For given platforms of 2 GPP & 1 RH tiles and 1 GPP & 1 RH tiles, the HDSE flow starts from tile count values of 3 and 2 respectively while evaluating heterogeneous tiles mappings by Algorithm 10, and evaluates a total of 31 and 25 mappings in a run-time of 19,683 and 15,825 ms, respectively. For given platforms containing tiles which are subset of total available implementation alternatives of actors in the application, HDSE flow discards evaluation of infeasible mappings requiring more tiles than available. Thus, DSE process gets speeded up in the case of smaller platforms.

7.4.4 Run-time Mapping Results

The results obtained from the hybrid run-time mapping (HRM) strategy proposed in this chapter have been compared with existing run-time strategies that start mapping an application without any previous analysis and perform the required analysis at run-time. Table 7.6 shows the time required (in milliseconds) to map throughput-constrained multimedia applications on a 4×4 MPSoC platform when the strategies Nearest Neighbor (NN) proposed in [129], Packing-based Best Neighbor (PBN), Communication-aware Packing-based Best Neighbor (CPBN) proposed in [J-1] and HRM are employed. The NN strategy tries to map the communicating actors on the same or neighboring tiles, whereas CPBN strategy tries to map the maximum communicating pairs of actors on the same tile. Then, throughput for the mapping is computed at run-time. Throughput computation for a mapping takes much more time than the time to find the mapping. The strategy needs to find a new mapping and then to calculate throughput for the same in case throughput-constraint is not fulfilled with the current mapping. Such strategies take time firstly in finding a mapping and secondly in computing throughput for the same,

	NN	PBN	CPBN	HRM
H.263 decoder	27.98	27.98	27.96	2.47
H.263 encoder	29.98	29.98	29.97	2.84
JPEG decoder	35.74	35.71	35.32	3.21
MP3 decoder	771.87	771.86	771.83	3.93

Table 7.6: Time required (in ms) to map the applications by different run-time mapping strategies

whereas the HRM strategy just selects the best mapping satisfying the throughput-constraint from the optimized mappings database. The selected mapping is used to configure the actors on the platform tiles. So, the total time consists of selection and placement time only. On average, HRM strategy is faster by about 93% as compared to CPBN that requires less time than NN and PBN.

7.4.5 Hop_distance Overestimation Penalty

Our DSE flows evaluate mappings by assuming that all the platform tiles are separated by some fixed hop_distance. However, in real situations, it is quite possible that not all available tiles at run-time are at the same hop_distance. Thus, our flow enforces a penalty for estimating higher hop_distances. At run-time, we look for a throughput satisfying mapping from the explored design-time mappings which contain tiles separated by maximum possible hop_distance between the available tiles. So, by mapping the actors on the available tiles based on the found mapping will definitely satisfy the throughput constraint since latency of some connections will be smaller as compared to ones considered during analysis. To map H.263 decoder on 4 ARM tiles, when all edges are mapped at a hop_distance of 2, i.e. tiles containing actors are separated by 2 hops, then throughput is 2.86168×10^{-6} (1/time-units) (Table 7.3) and when 2 edges are mapped at hop_distance of 1 and remaining edges at hop_distance of 2, then throughput is 2.86343×10^{-6} (1/time-units). The two throughput values vary only by 0.0006% and thus very less penalty in overestimating hop_distances. Therefore, the stored results from our design-time analysis are acceptable to be used for run-time mapping. Further, we always get better throughput than the stored one as actors are mapped on available tiles, making the approach suitable for real-time.

7.5 Summary

In this chapter, we have proposed a hybrid mapping strategy for efficient run-time mapping of throughput-constrained applications on MPSoC platforms. The hybrid strategy first performs extensive design-time analysis of the applications providing multiple design points. This is followed by a run-time mapping strategy to select the best point from the many available design points subject to available resources and desired throughput in order to map an application. In contrast, the existing mapping strategies perform mapping either at design-time or at run-time without any previous analysis of the application. A design-time strategy is incapable of handling dynamism in applications and a run-time approach can miss the timing deadline due to large computation requirements. The hybrid approach performs compute intensive analysis at design-time and leaves for minimal computation at run-time. This facilitates for a light-weight run-time platform manager that dynamically and efficiently configures the applications based on the platform resources status.

In order to perform design-time analysis of applications, we have proposed efficient design space exploration (DSE) strategies. The DSE strategies consider a generic MPSoC platform while performing design-time analysis, making the generated design points to be applicable to any target platform so long the target platform tile types and maximum separation between the tiles are subset of the tile types and maximum separation considered during DSE. It has been shown that our DSE strategies are scalable with the number of application tasks and platform tiles. Further, our DSE strategies are faster and provide better quality of solutions when compared to existing approaches. During the design-time analysis, an optimization is performed on the design points to discard non-optimal points, which results in reduced memory requirement to store them and facilitates for faster run-time selection. The newly proposed run-time mapping strategy is very efficient as it uses design-time analysis results, whereas conventional run-time approaches perform the time consuming analysis at run-time. However, the hybrid approach has limited flexibility, since all applications and maximum platform size must be known at design-time for performing the analysis.

Chapter 8

Conclusion and Future Directions

In this chapter, we summarize the contributions presented in this thesis and close with possible future research directions.

8.1 Conclusion

This thesis has proposed a number of novel techniques for run-time mapping of applications on NoC-based Heterogeneous MPSoC architectures. A detailed literature survey helped to establish that the mapping techniques were mostly limited to mapping one task per processing element (PE). It was also established that communication bottlenecks continue to be a concern, thereby limiting performance gains in large scale realizations.

In this thesis, a new packing strategy has been devised to minimize the communication overhead. Three new approaches based on the packing strategy were developed and evaluated in an attempt to improve the performance for the case of one task per PE based mapping process. The first approach aims to map tasks of an application in close proximity so as to reduce the communication overhead. In this approach, each task was mapped on the first available PE that is in closest proximity to the PE supporting its nearest neighbor. While this approach lends well for rapid mapping, it does not consider the best neighbor for a given set of tasks to be mapped at any given time. The second approach was extended to consider all free supported neighboring PEs to identify the best neighboring PE. This was shown to reduce the communication overhead compared to the first method. Moreover, it results in a more uniform distribution of channel loads be it at the expense of run-time overhead to find the best neighbor. This prompted the need to devise a third method to identify the most appropriate neighbor within a given time constraint. The proposed time bounded approach provides for a systematic method to identify the best possible neighbor without incurring unacceptable delays. It was clearly established that the time bounded approach is essential for applications with larger number of tasks. Our experimentation using varying number of tasks for a given MPSoC platform show that the overall performance is improved when compared to the second approach that identifies the best neighbor at all times. Moreover, comparisons with state-of-the-art techniques show that the proposed techniques lead to reduction in the average channel load of up to 22% for certain application scenarios.

The techniques that have been proposed for MPSoCs supporting only single task on each PE were subsequently extended to MPSoCs supporting multiple tasks on each PE. The extended techniques take advantage of multi-task supported PEs by mapping tightly-coupled communicating tasks on the same PE whenever possible in order to reduce the overall communication overhead. This has also contributed to significant reduction in the communication time and energy savings. When compared to MPSoC with single task supported PEs, the proposed approach leads to the identification of a better neighbor without increasing the time bound. In addition, the time required for computing the best neighbors have decreased notably.

Communication-aware mapping strategies were also introduced to favor PEs consisting of tasks that exhibit tighter communication with the task to be mapped. This has provided for a stronger clustering of tasks on a PE to further reduce the overall communication overhead. An accurate energy consumption model that estimates total energy as the sum of computation and communication energy has been proposed to evaluate the effect of this approach on energy consumption. Experimental evaluations show that the proposed techniques consistently lead to reduction in the average channel load, total execution time, average packet latency and energy consumption as these performance metrics depend directly on the communication overhead. In particular, energy savings of up to 46% along with improvement in other performance metrics can be realized. Furthermore, the total execution time can be reduced by up to 90% when compared to existing methods that do not consider communication-aware mapping of multiple tasks on a single PE.

A new computation and communication aware mapping technique has been proposed to provide for a more holistic approach to address both the computation and communication costs. This approach relies on the systematic elimination of the longest communication path till computation load on any PE becomes the bottleneck. This is followed by a process to merge tasks with minimum computation loads provided that the performance of the associated PE does not degrade the overall performance. Experimental results based on mapping models of real-life streaming applications with varying number of tasks show that considering both the computation and communication overheads leads to notable performance improvement when compared to approaches that considers only the communication overhead. In a case study to map multiple scenarios of an MPEG-4 application, we show that total execution time, energy consumption and resource usage are reduced by 33%, 39% and 37%, respectively.

A hybrid mapping strategy has also been proposed to accelerate the run-time mapping process in situations where the applications are known at design-time. It relies on extensive design-time analysis to derive multiple design points that can aid the run-time mapping process. This provides for a light-weight run-time mapping technique to select the best mapping solution given a set of tasks and the available PEs for mapping. The proposed hybrid strategy has been shown to accommodate a range of heterogeneous architectures. Experimental evaluations reveal that the proposed strategy speeds up the run-time mapping by 93%. Moreover, the time required for the design-time analysis can be reduced by 83% when compared to most recent method reported in the literature.

In conclusion, this thesis has presented novel run-time mapping techniques to handle dynamism in applications incurred at run-time. The techniques have shown great potential for streaming multimedia applications and have potential to perform well for different domain of applications that exhibit dynamism. It is noteworthy that the proposed techniques lend well to minimize the run-time mapping overhead without compromising the overall compute performance.

8.2 Future Research Directions

This section briefly discusses some future research directions which can extend or augment the work in this thesis.

- **Increasing Heterogeneity in Processing Elements**: The mapping techniques proposed in this thesis mainly consider two types of processing elements (PEs), general purpose processor (GPP) and reconfigurable hardware (RH). However, it would be interesting to extend this work to support the increase in the degree of heterogeneity for incorporating different types of PEs. This must be addressed with care due to the potential explosion in the number of permutations to be considered at each stage. Although, the proposed techniques can be extended for increased heterogeneity in the PEs, they will require to establish an upper limit on the heterogeneity in order to maintain the low complexity of the techniques.

- **Accelerating Design-time Analysis for Large Scale Problems**: The proposed design-time analysis techniques have been shown to be more efficient and faster when compared to the existing techniques. With the emerging trend to avail large number of PEs to increase the performance of large size applications containing hundreds of tasks, there exists a need to devise novel techniques to cater for large scale problems. The techniques will need to evaluate only efficient design points from a large number of possible design-points.

- **Realizing Run-time Mapping on FPGA-based Hardware Platforms**: Preliminary work has been carried out to successfully map real-life applications on NoC-based MPSoC on FPGA platform [C-5]. It would be interesting to evaluate the proposed run-time mapping techniques on such a platform. This provides for the detailed evaluation of communication and computation costs for each of the proposed methods. Also, the additional resources required and their implications on the compute performance can be examined in detail. It is worth exploring the possibility to accelerate the run-time mapping process in hardware.

- **Run-time Application-aware Architecture Morphing for Performance Improvement**: The mapping techniques target a fixed architecture at present. However, with the feasibility to implement a heterogeneous MPSoC platform on FPGA, the mapping techniques could be further extended to support the run-time customization of the MPSoC architecture so as to optimize the utilization of PEs. It is envisaged that the MPSoC architecture could be altered to support alternative mapping scenarios to satisfy both the functional and non-functional requirements. The run-time customization will include reconfiguring the FPGA fabric tiles for different type of PEs in order to achieve maximum performance. The performance will improve with increased heterogeneity in the architecture, but at the cost of reconfiguration overhead. Therefore, the extended techniques will need to maximize the performance while simultaneously minimizing the configuration overhead.

- **Run-time Faults Consideration for Fault Tolerance**: The run-time mapping techniques presented in this thesis are very efficient. However, they cannot cope with the random failures of PEs during run-time. Noting the reliability concerns of emerging VLSI technologies, it would be appropriate to explore ways to accommodate such scenarios. This could be in the form of devising strategies to rely on contingencies. Also, it is worth exploring ways to introduce sufficient redundancy in the PEs or devising techniques to maximize the harvest of available PEs through reconfiguration.

- **Extending Mapping Techniques to Consider Leakage Power**: It would be interesting to incorporate leakage awareness in the mapping strategies as leakage current is rapidly increasing with the shrinking device dimension. A five-fold increase in leakage current is predicted with each technology generation and thus recently there is large focus on the work on power and energy (leakage and dynamic energy) minimization. Recent works consider DVFS enabled processors for energy minimization. Therefore, the packing-based strategies can be extended in order to consider energy minimization in future research work.

The above research directions need to be carried out to take the run-time mapping techniques for MPSoCs into the next era.

References

[1] E. Gordon, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 4, pp. 114–117, 1965.

[2] "IMEC MPSoC Mapping Tools," 2008, http://www.imec.be/ScientificReport/SR2008/HTML/1225004.html (Last visited: 23 December, 2011).

[3] A. Jerraya, H. Tenhunen, and W. Wolf, "Guest Editors' Introduction: Multiprocessor Systems-on-Chips," *Computer*, vol. 38, no. 7, pp. 36–40, 2005.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[5] "The keys to success in multicore application development," 2009, http://eetimes.com/design/embedded/4008333/The-keys-to-success-in-multicore-application-development (Last visited: 23 December, 2011).

[6] P. Ross, "Why CPU Frequency Stalled," *Spectrum, IEEE*, vol. 45, no. 4, pp. 72–72, April 2008.

[7] "MPSoC Forum," http://www.mpsoc-forum.org/ (Last visited: 23 December, 2011).

[8] "Mapping Applications to MPSoCs," 2009, http://www.artist-embedded.org/artist/Overview,1614.html (Last visited: 23 December, 2011).

[9] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the Design Automation Conference*, 2007, pp. 746–749.

[10] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Citeseer, Tech. Rep., 2006.

[11] "INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS (ITRS), 2008 UPDATE," 2008, http://www.itrs.net/Links/2008ITRS/Home2008.htm (Last visited: 23 December, 2011).

[12] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, January 2002.

[13] J. Henkel, W. Wolf, and S. Chakradhar, "On-chip networks: A scalable, communication-centric embedded system design paradigm," in *Proceedings of the International Conference on VLSI Design*, 2004, pp. 845 – 851.

[14] D. Bertozzi and L. Benini, "Xpipes: a network-on-chip architecture for gigascale systems-on-chip," *IEEE Circuits and Systems Magazine*, vol. 4, no. 2, pp. 18–31, 2004.

[15] M. Hill and M. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, pp. 33 –38, July 2008.

[16] "Amdahl's Law Demonstration," http://en.wikipedia.org/wiki/Amdahl's_law (Last visited: 23 December, 2011).

[17] "Multi-core chips by academia and industry," http://en.wikipedia.org/wiki/Multi-core (Last visited: 23 December, 2011).

[18] "Raw Architecture Workstation (RAW)," 2003, http://groups.csail.mit.edu/cag/raw/ (Last visited: 23 December, 2011).

[19] "Asynchronous Array of Simple Processors (AsAP)," http://www.ece.ucdavis.edu/vcl/asap/ (Last visited: 23 December, 2011).

[20] "Tera-op, Reliable, Intelligently adaptive Processing System (TRIPS)," http://www.cs.utexas.edu/∼trips/index.html (Last visited: 23 December, 2011).

[21] "WaveScalar processor," 2006, http://wavescalar.cs.washington.edu/index.html (Last visited: 23 December, 2011).

[22] N. Saint-Jean, G. Sassatelli, P. Benoit, L. Torres, and M. Robert, "HS-Scale: a Hardware-Software Scalable MP-SOC Architecture for embedded Systems," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2007, pp. 21–28.

[23] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS," in *Proceedings of the IEEE International Solid-State Circuits Conference*, February 2007, pp. 98–589.

[24] "First 100-core Processor with the New TILE-Gx Family," 2009, http://www.tilera.com/ (Last visited: 23 December, 2011).

[25] S. Richardson, "MPOC: A chip multiprocessor for embedded systems," *HP Laboratories Technical Report HPL-2002-186, Palo Alto, CA, USA*, 2002.

[26] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, and H. Corporaal, "Run-time management of a MPSoC containing FPGA fabric tiles," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 16, pp. 24–33, January 2008.

[27] G. J. Smit, A. B. Kokkeler, P. T. Wolkotte, and M. D. van de Burgwal, "Multi-core architectures and streaming applications," in *Proceedings of the international workshop on System level interconnect prediction*, 2008, pp. 35–42.

[28] "4S - Smart ChipS for Smart Surroundings," 2007, http://caes.ewi.utwente.nl/research/8-projects/8 (Last visited: 23 December, 2011).

[29] T. Arpinen, P. Kukkala, E. Salminen, M. Hännikäinen, and T. D. Hämäläinen, "Configurable multiprocessor platform with RTOS for distributed execution of UML 2.0 designed applications," in *Proceedings of the conference on Design, automation and test in Europe*, 2006, pp. 1324–1329.

[30] N. Hristo, S. Todor *et al.*, "Automated Integration of Dedicated Hardwired IP Cores in Heterogeneous MPSoCs Designed with ESPAM," *EURASIP Journal on Embedded Systems*, vol. 2008, 2008.

[31] P. Heysters and G. Smit, "Mapping of DSP algorithms on the MONTIUM architecture," in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003, pp. 22 – 26.

[32] "Altera FPGAs," http://www.altera.com/ (Last visited: 23 December, 2011).

[33] A. Abnous, "Low-Power Domain Specific Processors for Digital Signal Processing," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2001. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2001/8190.html

[34] P. G. Paulin, C. Pilkington, E. Bensoudane, M. Langevin, and D. Lyonnard, "Application of a Multi-Processor SoC Platform to High-Speed Packet Forwarding," in *Proceedings of the conference on Design, automation and test in Europe*, 2004, pp. 58–63.

[35] J. Leijten, J. van Meerbergen, A. Timmer, and J. Jess, "PROPHID: a heterogeneous multi-processor architecture for multimedia," in *Proceedings of the International Conference on Computer Design*, 1997, pp. 164–169.

[36] M. J. Rutten, J. T. J. van Eijndhoven, E. G. T. Jaspers, P. van der Wolf, E.-J. D. Pol, O. P. Gangwal, and A. Timmer, "A Heterogeneous Multiprocessor Architecture for Flexible Media Processing," *IEEE Des. Test*, vol. 19, pp. 39–50, July 2002.

[37] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, 2006.

[38] S. Weiss and J. Smith, *POWER and PowerPC.* Morgan Kaufmann Publishers, 1994.

[39] J. De Oliveira and H. Van Antwerpen, "The Philips Nexperia digial video platforms," *Winning the SoC Revolution. Experiences in Real Design*, pp. 67–96, 2003.

[40] P. Cumming, "THE TI OMAP PLATFORM APPROACH TO SOC," *Winning the SoC revolution: experiences in real design*, 2003.

[41] A. Artieri, "Nomadik: an MPSoC Solution for Advanced Multimedia," in *Proceedings of the International Forum on Application-Specific Multi-Processor SoC (MPSoC)*, 2005.

[42] V. Baumgarte, G. Ehlers, F. May, A. Nuckel, M. Vorbach, and M. Weinhardt, "PACT XPPa self-reconfigurable data processing architecture," *Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[43] I. Held and B. VanderWiele, "Avispa-CH-embedded communications signal processor for multi-standard digital television," *GSPx TV to Mobile*, 2006.

[44] "Silicon Hive Inc," http://www.siliconhive.com/ (Last visited: 23 December, 2011).

[45] "Tensilica Inc," http://www.tensilica.com/ (Last visited: 23 December, 2011).

[46] R. E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, vol. 20, pp. 60–70, March 2000.

[47] T. Halfhill, "Busy Bees at Silicon Hive," *Microprocessor Report*, vol. 19, no. 6, pp. 17–20, 2005.

[48] C. Rowen, "Using configurable processors for high-efficiency multiple-processor systems," in *ERSA*, 2006, pp. 7–10.

[49] B. Hounsell and R. Taylor, "Co-Processor Synthesis: A New Methodology for Embedded Software Acceleration," in *Proceedings of the conference on Design, automation and test in Europe*, 2004, pp. 682 – 683.

[50] T. J. Callahan, "Automatic compilation of c for hybrid reconfigurable architectures," Ph.D. dissertation, 2002, chair-Wawrzynek, John.

[51] L. Smit, G. Smit, J. Hurink, H. Broersma, D. Paulusma, and P. Wolkotte, "Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture," in *Proceedings of International Conference on Field-Programmable Technology*, December 2004, pp. 421–424.

[52] S. Borgio, D. Bosisio, F. Ferrandi, M. Monchiero, M. D. Santambrogio, D. Sciuto, and A. Tumeo, “Hardware DWT accelerator for MultiProcessor System-on-Chip on FPGA,” in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2006, pp. 107–114.

[53] A. A. Jerraya and W. Wolf, “MultiProcessor Systems-on-Chips, Chapter The What, Why, and How of MPSoCs,” pp. 1–20, 2005.

[54] “Xilinx FPGAs,” http://www.xilinx.com/ (Last visited: 23 December, 2011).

[55] E. Salminen, A. Kulmala, and T. D. Hamalainen, “On network-on-chip comparison,” in *Proceedings of the Euromicro Conference on Digital System Design Architectures, Methods and Tools*, 2007, pp. 503–510.

[56] R. Marculescu, U. Ogras, L.-S. Peh, N. Jerger, and Y. Hoskote, “Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 1, pp. 3–21, January 2009.

[57] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. A. Zeferino, “SPIN: A Scalable, Packet Switched, On-Chip Micro-Network,” in *Proceedings of the conference on Design, Automation and Test in Europe*, 2003, pp. 70 – 73.

[58] K. Goossens, J. Dielissen, and A. Radulescu, “AEthereal Network on Chip: Concepts, Architectures, and Implementations,” *IEEE Des. Test*, vol. 22, no. 5, pp. 414–421, 2005.

[59] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, “QNoC: QoS architecture and design process for network on chip,” *J. Syst. Archit.*, vol. 50, pp. 105–128, February 2004.

[60] C. Hilton and B. Nelson, “PNoC: a flexible circuit-switched NoC for FPGA-based systems,” *IEE Proceedings on Computers and Digital Techniques*, vol. 153, no. 3, pp. 181 – 188, May 2006.

[61] D. Castells-Rufas, J. Joven, and J. Carrabina, “A Validation And Performance Evaluation Tool for ProtoNoC,” in *Proceedings of the International Symposium on System-on-Chip*, November 2006, pp. 1 – 4.

[62] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, “Guaranteed Bandwidth Using Looped Containers in Temporally Disjoint Networks within the Nostrum Network on Chip,” in *Proceedings of the conference on Design, automation and test in Europe*, 2004, pp. 890 – 895.

[63] T. Bjerregaard and S. Mahadevan, “A survey of research and practices of Network-on-chip,” *ACM Comput. Surv.*, vol. 38, no. 1, 2006.

[64] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, “HERMES: an infrastructure for low area overhead packet-switching networks on chip,” *Integr. VLSI J.*, vol. 38, no. 1, pp. 69–93, 2004.

[65] U. Y. Ogras and R. Marculescu, “Energy- and Performance-Driven NoC Communication Architecture Synthesis Using a Decomposition Approach,” in *Proceedings of the conference on Design, Automation and Test in Europe*, 2005, pp. 352–357.

[66] A. Chagoya-Garzon, X. Guerin, F. Rousseau, F. Petrot, D. Rossetti, A. Lonardo, P. Vicini, and P. S. Paolucci, “Synthesis of Communication Mechanisms for Multi-tile Systems Based on Heterogeneous Multi-processor System-On-Chips,” in *Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping*, 2009, pp. 48–54.

[67] Z. Yang, A. Kumar, and Y. Ha, “An area-efficient dynamically reconfigurable Spatial Division Multiplexing network-on-chip with static throughput guarantee,” in *Proceedings of the International Conference on Field-Programmable Technology*, December 2010, pp. 389 –392.

[68] “Sonics Inc: The Nettwork-on-Chip Company,” http://www.sonicsinc.com/ (Last visited: 23 December, 2011).

[69] “Arteris: The Nettwork-on-Chip Company,” http://www.arteris.com/ (Last visited: 23 December, 2011).

[70] J. Hu and R. Marculescu, "DyAD: smart routing for networks-on-chip," in *Proceedings of the Design Automation Conference*, 2004, pp. 260–263.

[71] S. Murali, D. Atienz, L. Benini, and G. De Michel, "A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 845–848.

[72] G. Martin, "Overview of the MPSoC design challenge," in *Proceedings of the Design Automation Conference*, 2006, pp. 274–279.

[73] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *J. VLSI Signal Process. Syst.*, vol. 41, no. 2, pp. 169–182, 2005.

[74] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "A Modular Approach to Model Heterogeneous MPSoC at Cycle Level," in *Proceedings of the EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, 2008, pp. 158–164.

[75] J. Cong, K. Gururaj, G. Han, A. Kaplan, M. Naik, and G. Reinman, "MC-Sim: an efficient simulation tool for MPSoC designs," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 364–371.

[76] Y. Atat and N.-E. Zergainoh, "Simulink-based MPSoC Design: New Approach to Bridge the Gap between Algorithm and Architecture Design," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2007, pp. 9–14.

[77] "SystemC," http://www.accellera.org/home/ (Last visited: 23 December, 2011).

[78] P. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: A System-Level Exploration Platform for Network Processors," *IEEE Des. Test*, vol. 19, pp. 17–26, November 2002.

[79] G. Beltrame, D. Sciuto, C. Silvano, P. Paulin, and E. Bensoudane, “An Application Mapping Methodology and Case Study for Multi-Processor On-Chip Architectures,” in *Proceedings of the IFIP International Conference on Very Large Scale Integration*, October 2006, pp. 146–151.

[80] H. Nikolov, T. Stefanov, and E. Deprettere, “Systematic and Automated Multiprocessor System Design, Programming, and Implementation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 542–555, March 2008.

[81] M. D. Nava, P. Blouet, P. Teninge, M. Coppola, T. Ben-Ismail, S. Picchiottino, and R. Wilson, “An Open Platform for Developing Multiprocessor SoCs,” *Computer*, vol. 38, no. 7, pp. 60–67, 2005.

[82] D. Atienza, P. G. Del Valle, G. Paci, F. Poletti, L. Benini, G. De Micheli, and J. M. Mendias, “A fast HW/SW FPGA-based thermal emulation framework for multiprocessor system-on-chip,” in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 618–623.

[83] F. Sun, S. Ravi, A. Raghunathan, and N. Jha, “Application-specific heterogeneous multiprocessor synthesis using extensible processors,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 9, pp. 1589–1602, September 2006.

[84] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal, “Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, pp. 40:1–40:27, July 2008.

[85] S. Lukovic and L. Fiorin, “An Automated Design Flow for NoC-based MPSoCs on FPGA,” in *Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping*, 2008, pp. 58–64.

[86] S. V. Tota, M. R. Casu, M. R. Roch, L. Macchiarulo, and M. Zamboni, “A case study for NoC-based homogeneous MPSoC architectures,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, pp. 384–388, March 2009.

[87] A. Kumar, A. Hansson, J. Huisken, and H. Corporaal, “Interactive presentation: An FPGA design flow for reconfigurable network-based multi-processor systems on chip,” in *Proceedings of the conference on Design, automation and test in Europe*, 2007, pp. 117–122.

[88] T. Dorta, J. Jimenez, J. Martin, U. Bidarte, and A. Astarloa, “Overview of FPGA-Based Multiprocessor Systems,” in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, December 2009, pp. 273 –278.

[89] G.-G. Mplemenos and I. Papaefstathiou, “MPLEM: An 80-processor FPGA Based Multiprocessor System,” in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, April 2008, pp. 273 –274.

[90] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz, “RAMP Blue: A Message-Passing Manycore System in FPGAs,” in *Proceedings of the International Conference on Field Programmable Logic and Applications*, August 2007, pp. 54 –61.

[91] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, “MAPS: an integrated framework for MPSoC application parallelization,” in *Proceedings of the Design Automation Conference*, 2008, pp. 754–759.

[92] R. P. Dick, D. L. Rhodes, and W. Wolf, “TGFF: task graphs for free,” in *Proceedings of the international workshop on Hardware/software codesign*, 1998, pp. 97–101.

[93] S. Stuijk, M. Geilen, and T. Basten, “SDF^3: SDF For Free,” in *Proceeding International Conference on Application of Concurrency to System Design*, June 2006, pp. 276–278.

[94] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano, “Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip,” in *Proceedings of the conference on Design, automation and test in Europe*, 2006, pp. 3–8.

[95] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, and L. Benini, "A fast and accurate technique for mapping parallel applications on stream-oriented MPSoC platforms with communication awareness," *Int. J. Parallel Program.*, vol. 36, no. 1, pp. 3–36, 2008.

[96] J. Hu and R. Marculescu, "Energy- and performance-aware mapping for regular NoC architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 4, pp. 551–562, April 2005.

[97] C. Marcon, A. Borin, A. Susin, L. Carro, and F. Wagner, "Time and energy efficient mapping of embedded applications onto NoCs," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2005, pp. 33–38.

[98] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli, "A methodology for mapping multiple use-cases onto networks on chips," in *Proceedings of the conference on Design, automation and test in Europe*, 2006, pp. 118–123.

[99] C.-E. Rhee, H.-Y. Jeong, and S. Ha, "Many-to-Many Core-Switch Mapping in 2-D Mesh NoC Architectures," in *Proceedings of the IEEE International Conference on Computer Design*, 2004, pp. 438–443.

[100] T. Lei and S. Kumar, "Algorithms and Tools for Network on Chip Based System Design," in *Proceedings of the 16th symposium on Integrated circuits and systems design*, 2003, p. 163.

[101] D. Wu, B. M. Al-Hashimi, and P. Eles, "Scheduling and Mapping of Conditional Task Graphs for the Synthesis of Low Power Embedded Systems," in *Proceedings of the conference on Design, Automation and Test in Europe*, 2003, p. 10090.

[102] S. Manolache, P. Eles, and Z. Peng, "Fault and energy-aware communication mapping with guaranteed latency for applications implemented on NoC," in *Proceedings of the Design Automation Conference*, 2005, pp. 266–269.

[103] L.-Y. Lin, C.-Y. Wang, P.-J. Huang, C.-C. Chou, and J.-Y. Jou, "Communication-driven task binding for multiprocessor with latency insensitive network-on-chip,"

in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2005, pp. 39–44.

[104] H. Orsila, T. Kangas, E. Salminen, T. D. Hämäläinen, and M. Hännikäinen, “Automated memory-aware application distribution for Multi-processor System-on-Chips,” *J. Syst. Archit.*, vol. 53, no. 11, pp. 795–815, 2007.

[105] M. Branca, L. Camerini, F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo, “Evolutionary algorithms for the mapping of pipelined applications onto heterogeneous embedded systems,” in *Proceedings of the Annual conference on Genetic and evolutionary computation*, 2009, pp. 1435–1442.

[106] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, “Mapping Applications to Tiled Multiprocessor Embedded Systems,” in *Proceedings of the International Conference on Application of Concurrency to System Design*, 2007, pp. 29–40.

[107] G. Chen, F. Li, S. Son, and M. Kandemir, “Application mapping for chip multiprocessors,” in *Proceedings of the ACM/IEEE Design Automation Conference*, June 2008, pp. 620–625.

[108] N. Satish, K. Ravindran, and K. Keutzer, “A decomposition-based constraint optimization approach for statically scheduling task graphs with communication delays to multiprocessors,” in *Proceedings of the conference on Design, automation and test in Europe*, 2007, pp. 57–62.

[109] P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, and L. Huang, “Mapping of applications to MPSoCs,” in *Proceedings of the IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2011, pp. 109–118.

[110] C.-L. Chou and R. Marculescu, “User-aware dynamic task allocation in networks-on-chip,” in *Proceedings of the conference on Design, automation and test in Europe*, 2008, pp. 1232–1237.

[111] Z. Peter, S. Gilles, U. Nurten, S. Nicolas, B. Pascal, and G. Manfred, "A Decentralised Task Mapping Approach for Homogeneous Multiprocessor Network-On-Chips," *International Journal of Reconfigurable Computing*, vol. 2009, 2009.

[112] E. W. Briáo, D. Barcelos, and F. R. Wagner, "Dynamic task allocation strategies in MPSoC for soft real-time applications," in *Proceedings of the conference on Design, automation and test in Europe*, 2008, pp. 1386–1389.

[113] C.-L. Chou and R. Marculescu, "Incremental run-time application mapping for homogeneous NoCs with multiple voltage levels," in *Proceedings of the IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2007, pp. 161–166.

[114] A. Ngouanga, G. Sassatelli, L. Torres, T. Gil, A. Soares, and A. Susin, "A contextual resources use: a proof of concept through the APACHES' platform," in *Proceedings of the IEEE Design and Diagnostics of Electronic Circuits and systems*, 2006, pp. 42–47.

[115] A. Mehran, A. Khademzadeh, and S. Saeidi, "DSM: A Heuristic Dynamic Spiral Mapping algorithm for network on chip," *IEICE Electronics Express*, vol. 5, no. 13, pp. 464–471, 2008.

[116] G. Sassatelli, N. Saint-Jean, P. Benoit, L. Torres, M. Robert, C. Woszezenki, I. A. Grehs, and F. Moraes, "Run-time mapping and communication strategies for Homogeneous NoC-Based MPSoCs," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 295–296.

[117] C. Ykman-Couvreur, V. Nollet, F. Catthoor, and H. Corporaal, "Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management," in *Proceedings of the International Symposium on System-on-Chip*, November 2006, pp. 1 –4.

[118] H. Shojaei, A. Ghamarian, T. Basten, M. Geilen, S. Stuijk, and R. Hoes, "A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for CMP run-time management," in *Proceedings of the 46th Annual Design Automation Conference*, 2009, pp. 917–922.

[119] O. Moreira, J. J.-D. Mol, and M. Bekooij, "Online resource management in a multiprocessor with a network-on-chip," in *Proceedings of the ACM symposium on Applied computing*, 2007, pp. 1557–1564.

[120] P. K. F. Hölzenspies, J. L. Hurink, J. Kuper, and G. J. M. Smit, "Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (MPSoC)," in *Proceedings of the conference on Design, automation and test in Europe*, 2008, pp. 212–217.

[121] T. D. ter Braak, P. K. F. Hölzenspies, J. Kuper, J. L. Hurink, and G. J. M. Smit, "Run-time spatial resource management for real-time applications on heterogeneous MPSoCs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 357–362.

[122] M. A. Al Faruque, R. Krist, and J. Henkel, "ADAM: run-time agent-based distributed application mapping for on-chip communication," in *Proceedings of the Design Automation Conference*, 2008, pp. 760–765.

[123] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Power-Aware Mapping of Probabilistic Applications onto Heterogeneous MPSoC Platforms," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009, pp. 151–160.

[124] T. Lei and S. Kumar, "A Two-step Genetic Algorithm for Mapping Task Graphs to a Network on Chip Architecture," in *Proceedings of the Euromicro Symposium on Digital Systems Design*, 2003, pp. 180 – 187.

[125] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the conference on Computing frontiers*, 2006, pp. 29–40.

[126] T. Theocharides, M. K. Michael, M. Polycarpou, and A. Dingankar, "Towards embedded runtime system level optimization for MPSoCs: on-chip task allocation," in *Proceedings of the ACM Great Lakes symposium on VLSI*, 2009, pp. 121–124.

[127] S. Schneider, A. Meisel, and W. Hardt, “Communication-Aware Hierarchical Online-Placement in Heterogeneous Reconfigurable Systems,” in *Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping*, 2009, pp. 61–67.

[128] J. Huang, A. Raabe, C. Buckl, and A. Knoll, “A workflow for runtime adaptive task allocation on heterogeneous MPSoCs,” in *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, March 2011, pp. 1 –6.

[129] E. Carvalho and F. Moraes, “Congestion-aware task mapping in heterogeneous MPSoCs,” in *Proceedings of the International Symposium on System-on-Chip*, November 2008, pp. 1–4.

[130] V. Nollet, T. Marescaux, P. Avasare, and J.-Y. Mignolet, “Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles,” in *Proceedings of the conference on Design, Automation and Test in Europe*, 2005, pp. 234–239.

[131] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, “Supporting task migration in multi-processor systems-on-chip: a feasibility study,” in *Proceedings of the conference on Design, automation and test in Europe*, 2006, pp. 15–20.

[132] H. Kalte and M. Porrmann, “Context saving and restoring for multitasking in reconfigurable systems,” in *Proceedings of the International Conference on Field Programmable Logic and Applications*, August 2005, pp. 223–228.

[133] O. Moreira, F. Valente, and M. Bekooij, “Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor,” in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, 2007, pp. 57–66.

[134] Y. Ahn, K. Han, G. Lee, H. Song, J. Yoo, K. Choi, and X. Feng, “SoCDAL: System-on-chip design AcceLerator,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, pp. 17:1–17:38, February 2008.

[135] J. Keinert, M. Streub&uhorbar;hr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SystemCoDesigner an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, pp. 1:1–1:23, January 2009.

[136] W. Liu, M. Yuan, X. He, Z. Gu, and X. Liu, "Efficient SAT-Based Mapping and Scheduling of Homogeneous Synchronous Dataflow Graphs for Throughput Optimization," in *Proceedings of the Real-Time Systems Symposium*, 2008, pp. 492–504.

[137] A. Bonfietti, M. Lombardi, M. Milano, and L. Benini, "Throughput Constraint for Synchronous Data Flow Graphs," in *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2009, pp. 26–40.

[138] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proceedings of the Design Automation Conference*, 2007, pp. 777–782.

[139] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria, "An industrial design space exploration framework for supporting run-time resource management on multi-core systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 196–201.

[140] N. H. Zamora, X. Hu, and R. Marculescu, "System-level performance/power analysis for platform-based design of multimedia applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, pp. 2:1–2:29, February 2007.

[141] S. Stuijk, M. Geilen, and T. Basten, "A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour," in *Proceedings of the Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010, pp. 548–555.

[142] B. Giovanni, L. Fossati, and D. Sciuto, "Decision-theoretic design space exploration of multiprocessor platforms," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 29, pp. 1083–1095, July 2010.

[143] G. Palermo, C. Silvano, and V. Zaccaria, “Multi-objective design space exploration of embedded systems,” *J. Embedded Comput.*, vol. 1, pp. 305–316, August 2005.

[144] G. Ascia, V. Catania, A. G. Di Nuovo, M. Palesi, and D. Patti, “Efficient design space exploration for application specific systems-on-a-chip,” *J. Syst. Archit.*, vol. 53, pp. 733–750, October 2007.

[145] M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich, “Efficient symbolic multi-objective design space exploration,” in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2008, pp. 691–696.

[146] Z. J. Jia, A. Pimentel, M. Thompson, T. Bautista, and A. Nunez, “NASA: A generic infrastructure for system-level MP-SoC design space exploration,” in *Proceedings of the IEEE Workshop on Embedded Systems for Real-Time Multimedia*, October 2010, pp. 41 –50.

[147] P. van Stralen and A. Pimentel, “Scenario-based design space exploration of MP-SoCs,” in *Proceedings of the IEEE International Conference on Computer Design*, October 2010, pp. 305 –312.

[148] G. Palermo, C. Silvano, and V. Zaccaria, “Robust optimization of SoC architectures: A multi-scenario approach,” in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, October 2008, pp. 7 –12.

[149] L. Benini, D. Bertozzi, and M. Milano, “Resource Management Policy Handling Multiple Use-Cases in MPSoC Platforms Using Constraint Programming,” in *Proceedings of the 24th International Conference on Logic Programming*, 2008, pp. 470–484.

[150] A. Schranzhofer, J.-J. Chen, and L. Thiele, “Dynamic Power-Aware Mapping of Applications onto Heterogeneous MPSoC Platforms,” *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 692 –707, November 2010.

[151] C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria, “Linking run-time resource management of embedded multi-core platforms

with automated design-time exploration," *IET Computers Digital Techniques*, vol. 5, no. 2, pp. 123 –135, March 2011.

[152] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins, "Managing dynamic concurrent tasks in embedded real-time multimedia systems," in *Proceedings of the 15th international symposium on System Synthesis*, 2002, pp. 112–119.

[153] L. Xue, O. ozturk, F. Li, M. Kandemir, and I. Kolcu, "Dynamic partitioning of processing and memory resources in embedded MPSoC architectures," in *Proceedings of the conference on Design, automation and test in Europe*, 2006, pp. 690–695.

[154] L. Möller, I. Grehs, E. Carvalho, R. Soares, N. Calazans, and F. Moraes, "A NoC-based Infrastructure to Enable Dynamic Self Reconfigurable Systems," pp. 1–27, 2007.

[155] H. Yu, Y. Ha, and B. Veeravalli, "Communication-aware application mapping and scheduling for NoC-based MPSoCs," in *Proceedings of International Symposium on Circuits and Systems*, 2010, pp. 3232 –3235.

[156] J. C. S. Palma et al., "Mapping embedded systems onto NoCs: the traffic effect on dynamic energy estimation," in *Proceedings of the Integrated circuits and system design*, 2005, pp. 196–201.

[157] J. Hu and R. Marculescu, "Energy-Aware Communication and Task Scheduling for Network-on-Chip Architectures under Real-Time Constraints," in *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, ser. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 10 234–. [Online]. Available: http://dl.acm.org/citation.cfm?id=968878.969037

[158] "TILE64 PROCESSOR," 2008, http://www.tilera.com/products/TILE64.php.

[159] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, "NoC Synthesis Flow for Customized Domain Specific Multiprocessor Systems-on-Chip," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, pp. 113–129, February 2005.

[160] C. Grecu, P. Pande, A. Ivanov, and R. Saleh, “Timing analysis of network on chip architectures for MP-SoC platforms,” *Microelectronics J.*, vol. 36, no. 9, pp. 833 – 845, 2005.

[161] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Trans. Comput.*, vol. 36, pp. 24–35, January 1987.

[162] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij, “Throughput Analysis of Synchronous Data Flow Graphs,” in *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, 2006, pp. 25–36.

[163] M. Geilen, T. Basten, B. Theelen, and R. Otten, “An algebra of Pareto points,” in *Proceedings of the International Conference on Application of Concurrency to System Design*, june 2005, pp. 88 – 97.

[164] S. Segars, “ARM7TDMI power consumption,” *IEEE Micro*, vol. 17, no. 4, pp. 12 –19, jul/aug 1997.

[165] S. H. Cho, T. Xanthopoulos, and A. Chandrakasan, “A low power variable length decoder for MPEG-2 based on nonuniform fine-grain table partitioning,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 2, pp. 249 –257, june 1999.

[166] M. Hentati, Y. Aoudni, J. Nezan, M. Abid, and O. Deforges, “FPGA dynamic reconfiguration using the RVC technology: Inverse quantization case study,” in *International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, nov. 2011, pp. 1 –7.

[167] T.-Y. Sung, Y.-S. Shieh, C.-W. Yu, and H.-C. Hsin, “High-Efficiency and Low-Power Architectures for 2-D DCT and IDCT Based on CORDIC Rotation,” in *International Conference on Parallel and Distributed Computing, Applications and Technologies*, dec. 2006, pp. 191 –196.

[168] J. Ren and N. Kehtarnavaz, “Comparison of Power Consumption for Motion Compensation and Deblocking Filters in High Definition Video Coding,” in *IEEE International Symposium on Consumer Electronics*, june 2007, pp. 1 –5.

[169] “Encyclopedia of Integer Sequences,” http://oeis.org/ (Last visited: 23 December, 2011).