

**MAPPING APPLICATIONS
ONTO CLUSTER-BASED
MPSOCS**

OLIVER BELLAVER LONGHI

Dissertation presented as partial requirement
for obtaining the degree of Master in
Computer Science at Pontifical Catholic
University of Rio Grande do Sul.

Advisor: Prof. Fabiano Passuelo Hessel

Dados Internacionais de Catalogação na Publicação (CIP)

L854m Longhi, Oliver Bellaver
Mapping applications onto cluster-based MPSOCS / Oliver
Bellaver Longhi. – Porto Alegre, 2014.
73 p.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Fabiano Passuelo Hessel.

1. Informática. 2. Arquitetura de Computador.
3. Mutiprocessadores. I. Hessel, Fabiano Passuelo. III. Título.

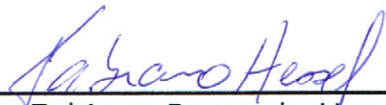
CDD 004.35

**Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS**

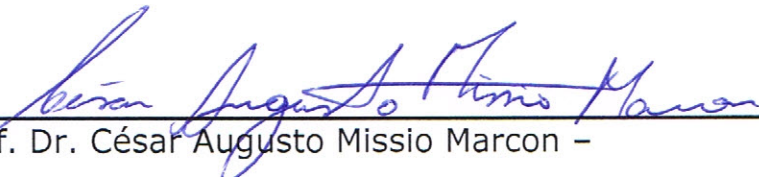


TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

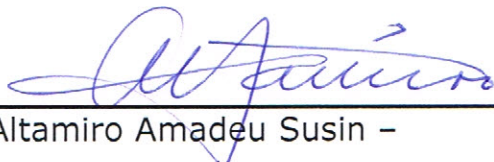
Dissertação intitulada "*Mapping Applications onto Cluster-based MPSoCs*" apresentada por Oliver Bellaver Longhi como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, aprovada em 12/03/2014 pela Comissão Examinadora:



Prof. Dr. Fabiano Passuelo Hessel - PPGCC/PUCRS
Orientador




Prof. Dr. César Augusto Missio Marcon - PPGCC/PUCRS



Prof. Dr. Altamiro Amadeu Susin - UFRGS

Homologada em 03/04/2014, conforme Ata No. 005 pela Comissão Coordenadora.



Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

To my family and friends.

“Ne frustra vixisse videar”
(Tycho Brahe)

ACKNOWLEDGMENTS

I am thankful for the help given by my coworkers Sérgio Johann, Felipe Magalhães and Carlos Moratelli, by my official and non-official advisors, respectively, Fabiano Hessel and Gabriel Marchesan de Almeida, and finally, by the enthusiastic professors César Marcon and Beatriz Franciosi.

MAPEAMENTO DE APLICAÇÕES EM MPSOCS CLUSTERIZADOS

RESUMO

Durante décadas, a indústria aumentava a frequência de operação dos processores para responder às necessidades de desempenho. Após atingir uma limitação física em termos de geração de calor, o novo eixo escolhido para explorar desempenho foi escalar o número de elementos de processamento. Para lidar com o crescente número de elementos de processamento, cada vez mais são importantes as metodologias para auxiliar os projetistas no desenvolvimento de sistemas multiprocessados. Abordagens baseadas em simulação e prototipação em FPGA são onerosas pois demandam muitos recursos, tais como projetistas e tempo. Por isso, técnicas baseadas em modelos analíticos ganham visibilidade como alternativas para essas abordagens onerosas. Porém, modelos analíticos possuem desvantagens, como a dificuldade de modelar e caracterizar diferentes arquiteturas. Além disso, topologias emergentes de sistemas multiprocessados carecem de modelos analíticos. Levando esse cenário em conta, este trabalho propõe um modelo analítico que suporta atividades comuns de projetistas tais como mapeamento de aplicações e geração de protótipos de sistemas multiprocessados.

Palavras Chave: Mapeamento de Tarefas, Particionamento de Tarefas, Redes Intra-chip, *clusters*.

MAPPING APPLICATIONS ONTO CLUSTER-BASED MPSoCs

ABSTRACT

The industry for decades has increased the clock rate to answer the need of performance. Reaching a physical limitations in terms of heat, the new chosen axis to increase performance is to scale the number of processing elements. To deal with that scaling number of processing elements, more and more important are the methodologies to support the design of MPSoCs. Approaches like simulation and FPGA-based prototyping are too expensive and timing consuming. Therefore, techniques like Analytical Models represent important alternatives to the previous consuming approaches. However, these architecture models are difficult to build and characterize. In addition, emerging MPSoC topologies lack analytical models. Due to that, this work proposes an analytical model to support designers in common tasks of the design process like application mapping and prototypes generation.

Keywords: Task mapping, Task partition, NoC, cluster.

LIST OF FIGURES

2.1	Flow showing how to apply partitioning and mapping. Adapted and translated from [Mar05].	25
2.2	Point-to-point infrastructure with sixteen processing elements. See that not every link direction is implemented.	28
2.3	Bus connecting 64 processing elements.	29
2.4	8x8 mesh-based NoC interconnecting 64 processing elements.	30
2.5	4x4x4 mesh-based NoC.	32
4.1	Flow to map applications to target architectures.	40
4.2	Different topologies composed by Routers, Processors and Channels.	42
4.3	Description of the cost function.	43
4.4	Random mapping algorithm.	44
4.5	Traditional implementation of the Simulated Annealing.	45
4.6	Second implementation of the Simulated Annealing.	46
4.7	Third implementation of the Simulated Annealing.	46
4.8	Bus specification.	49
4.9	Router specification.	50
4.10	CNoC Local Interface Structure.	51
4.11	Communication protocol header.	52
4.12	Target flit specification.	53
4.13	XYCSIM help command.	54
4.14	XYCMAP help command.	55
4.15	XYCADAPT help command.	56
4.16	XYCPRO help command.	57
5.1	<i>Send</i> task description.	60
5.2	Performance analysis of application 1.	64
5.3	Performance analysis of application 2.	65
5.4	Performance analysis of applications 1 and 2.	65
5.5	Mapping costs for the 4×4 architecture.	67
5.6	Mapping costs for the $2 \times 2 \times 4$ architecture.	67

LIST OF TABLES

3.1	Comparison of cluster-based works.	37
4.1	Hellfire communication primitives.	51
5.1	Bus (64) time costs.	60
5.2	NoC (8x8) time costs.	61
5.3	Clustered NoC (2x8x4) time costs.	62
5.4	Area comparison between NoC and clustered NoC.	62
5.5	Frequency comparison between NoC and clustered NoC on Virtex V. Adapted from [Mag13].	63

LIST OF ACRONYMS

GSE – Embedded Systems Group
ITIV – Institute for Information Processing Technologies
PUCRS – Pontifical Catholic University Of Rio Grande Do Sul
KIT – Karlsruhe Institute of Technology
PE – Processing Element
CPU – Central processing unit
ITRS – International Roadmap for Semiconductors
FPGA – Field-Programmable Gate Array
SOC – System-on-Chip
RM – Rate Monotonic
EDF – Earliest Deadline First
DSP – Digital Signal Processing
MPSOC – Multiprocessor System-on-Chip
NOC – Network-on-chip
CNOC – Cluster-based Network-on-Chip
OVP – Open Virtual Platforms
API – Application Programmable Interface
MCWG – Modified Communication-Weighted Graph
SA – Simulated Annealing
SAN – Nested Simulated Annealing
HDL – Hardware Description Language
HFOS – Hellfire Operating System
HC-MPSOC – Hellfire Cluster-based MPSoC

CONTENTS

1	INTRODUCTION	23
1.1	ORGANIZATION	24
2	RELATED CONCEPTS	25
2.1	PARTITIONING AND MAPPING TASKS	25
2.1.1	PARTITIONING	25
2.1.2	MAPPING	26
2.1.3	COMBINATION OF TECHNIQUES	27
2.2	COMMUNICATION STRUCTURES	27
2.2.1	POINT-TO-POINT	27
2.2.2	BUS	28
2.2.3	NETWORK-ON-CHIP	30
2.2.4	CLUSTERED NOC	32
3	RELATED WORKS	35
3.1	MAPPING TECHNIQUES FOR NETWORK-ON-CHIP	35
3.2	MAPPING TECHNIQUES FOR CLUSTER-BASED MPSOCS	36
3.3	DISTINCTION OF PROPOSED WORK	37
4	PROPOSED WORK	39
4.1	ADOPTED MODELS AND THE MAPPING STEP	40
4.1.1	COMMUNICATION TASK GRAPHS	40
4.1.2	ABSTRACT ARCHITECTURES	41
4.1.3	FUNCTION COST	42
4.1.4	MAPPING ALGORITHMS	43
4.2	HELLFIRE SYSTEM	47
4.2.1	IMPLEMENTED ARCHITECTURES	48
4.2.2	COMMUNICATION DRIVER	50
4.3	XYC TOOLCHAIN	53
4.3.1	XYCSIM - XYC SIMULATOR	53
4.3.2	XYCMAP - XYC MAPPER	54
4.3.3	XYCADAPT - XYC ADAPTOR	56
4.3.4	XYCPRO - XYC PROTOTYPE BUILDER	56

5	EXPERIMENTS	59
5.1	ARCHITECTURE ANALYSIS	59
5.1.1	OTHER CONSTRAINTS	62
5.2	SYSTEM SIMULATION	63
5.3	MAPPING ALGORITHMS	66
6	FINAL CONSIDERATIONS	69
	REFERENCES	71

1. INTRODUCTION

The industry for decades has increased the clock rate to answer the need of performance. Reaching a physical limitation in terms of heat dissipation, the new chosen axis to increase performance is to scale the number of processing elements (PE). As a consequence, ITRS (International Roadmap for Semiconductors) expects System-on-Chips (SoC) to contain thousands of processing elements near to 2020 [ITR09]. Actually, now it is already possible to find SoCs that reach the hundreds of PEs as expected [KJS⁺02] in 2002.

Such systems that contain multiple processing elements are known as MPSoCs (Multi-processor System-on-Chip). Usually, the processing elements are interconnected to each other by an on-chip interconnection infrastructure. To compose such interconnection, different approaches might be used. Few approaches use dedicated bus, others use shared bus, and there are even others that use a more complex communication media, such as Network-on-Chip (NoC), that involves the use of routers.

In a more formal manner, NoCs can be defined as intra-chips communication infrastructures, usually composed by a set of routers interconnected by point to point communication channels, implementing a chosen topology. Their main advantages are high scalability, reusability and reliability [DT01]. These characteristics make NoCs good candidates for current and future MPSoCs designs.

A novel approach that explores better domain-specific architectures has been studied recently [BFFM12] [MBF⁺12] [FYX⁺10] [TMT12]. They are known as cluster-based MPSoCs, and differently from NoCs that compose links to processing elements, they compose links to a second-level communication infrastructure, usually a bus.

However, as more and more sophisticated software can be executed on MPSoCs, and as the complexity of software to be developed is increasing, the developers can no longer wait for the chip to be fabricated or prototyped for the integration of hard/software phase in order to meet the time-to-market constraints [HYH⁺11]. To deal with that, a set of tools are required to keep different specialists working in their respective fields. It means that developers and designers need to work at the same project in different models using different approaches.

The two main approaches to develop and design MPSoCs are simulation-based and FPGA-based. The first one uses high-level models of processing components to assume their expected behaviour. The second can be used to prototype system and perform test and validation. Both of them have their pros and cons. Simulation-based tools have intrinsic sequential nature and their execution time grows linearly as the number of processing elements grows. FPGA-based can achieve high execution times but are more complex and, again, this complexity grows according to the number of processing elements. In addition, FPGA-based validation requires investment and they use to be expensive.

As the cost and time to prototype and to simulate MPSoCs increase, more important are the design-time methods to build efficient SoCs. These methods can help designers to estimate several requisites of systems, like energy consumption, memory and performance. But, the most important, they base designers' decisions aiming to better exploit resource and design space. Methods like these are labelled as analytical models, and as FPGA-based and simulation-based methods, it has its drawbacks. They are harder to model and are strongly coupled to specific models, what requires a new effort to adapt new technology to the analytical model.

A special case where analytical models are employed are the tasks of mapping and partitioning applications to physical network structure. These tasks are important because, depending on the mapping of communicating tasks, the bandwidth might increase, and by allocating higher bandwidth across the links of the NoC, less performance is obtained and more energy is dissipated. Thus, it is important to balance the bandwidth needs across the different links [MDM04].

After glimpsing this whole scenario, it is possible to realize that debugging, validating and verifying are ever growing and challenging issues in the embedded systems field. It means that, taking into consideration that systems are facing growth of complexity with respect to the number of processing elements being used, the effort on the development of tools to deal with these issues is very important. In addition, the existing tools are powerful to deal with specific levels of abstraction, but the ones that work with different types of methods are more difficult to build and comprehend much more complexity. At that point, a tool that uses simulations and analytical models could explore others gaps of MPSoCs design. Therefore, this work presents and emphasizes the need of tools and methods that ease the design space exploration with respect to models that use a combination of methods (analytical models and simulation-based tools).

The objective of this work is to propose a new model to map applications to cluster-based networks. To do that, the HC-MPSoC [Mag13], a cluster-based network similar to [BFFM12] and [MBF⁺12] that has been designed in GSE (Embedded Systems Groups), is going to be the target infrastructure of this work. The constraints explored by the work are the number of cores, area, and the latency. The number of cores is an important constraint because as less PEs are in the project less area is needed and, therefore, the overall frequency of the architecture can be increased, improving the system performance [TMT12]. Thus, a mapping approach to search for the best form factor to build MPSoCs is also presented as an achievement of the work.

1.1 Organization

In Chapter 2 several important concepts for the understanding of the work are presented. Chapter 3 collects few works that employ studies about popular analytical models, mapping/partitioning algorithms and cluster-based MPSoCs. After that, Chapter 4 presents the proposed work. In Chapter 5 a set of experiments that support the work realized are presented. Finally, in Chapter 6 some considerations are made and future works are proposed.

2. RELATED CONCEPTS

2.1 Partitioning and Mapping Tasks

Partitioning and mapping tasks are important concepts to this work. Figure 2.1 demonstrate in a graphical manner how both techniques are related. In the partitioning process, tasks are arranged in blocks. These blocks are posteriorly placed in defined tiles of the available infrastructure. Tiles are pairs of resources and its access points. We can consider resources as processors, peripherals and memories while access points can be network interfaces or channels that link processing elements to the communication infrastructure.

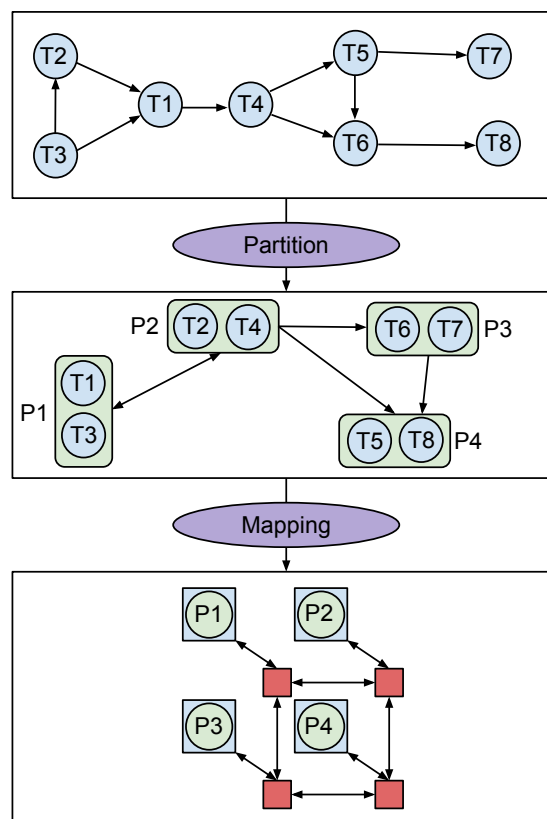


Figure 2.1: Flow showing how to apply partitioning and mapping. Adapted and translated from [Mar05].

2.1.1 Partitioning

The partitioning step takes a set of tasks as input and gives a set of blocks that contain the task set. Each resulting block has to respect one or more constraints. One common type of constraint is scheduling policies. Basically, it ensures that the task set mapped to the block can be scheduled by the operating system without overcharging the processing element. Scheduling

policies can be very simple as round-robin or FIFO, or be more complex like RM (Rate Monotonic) and EDF (Earliest Deadline First). Additionally, there are other constraints, e.g., one task set should be mandatorily mapped to predefined blocks or blocks should not exceed a given number of tasks. Actually, these constraints may vary with design requisites and resources available.

Let $\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \dots, \gamma_n\}$ be the set of existing tasks of the system, $\beta \subseteq \Gamma$ a subset from Γ known as block and $\mathbf{B} = \{\beta_1, \beta_2, \beta_3, \dots, \beta_m\}$ a set of all blocks. Then, the partitioning is a process that originates \mathbf{B} , considering that $\beta_1 \cup \beta_2 \cup \beta_3 \cup \dots \cup \beta_m = \Gamma$, $m = |\Gamma|$ and $\beta_k \cap \beta_i = \emptyset$ given that $\forall_{\beta_k, \beta_i} k \neq i$.

The biggest β possible is the one composed by all tasks in Γ . In this case, the existing block is the system itself. On the other side, the smallest β of a partition \mathbf{B} is the one that contains a single γ . This case represents a partition whose $n = m$.

The complexity to perform the partitioning of tasks to blocks is proportional to the Bell number $O(\text{Bell}(n))$, where n is the size of the task set. For instance, a set Γ of size $n = 3$ has the following acceptable combination of blocks:

$$\begin{aligned} & \{\{\gamma_1\}, \{\gamma_2\}, \{\gamma_3\}\} \\ & \{\{\gamma_1\}, \{\gamma_2, \gamma_3\}\} \\ & \{\{\gamma_2\}, \{\gamma_1, \gamma_3\}\} \\ & \{\{\gamma_3\}, \{\gamma_1, \gamma_2\}\} \\ & \{\{\gamma_1, \gamma_2, \gamma_3\}\} \end{aligned}$$

2.1.2 Mapping

The mapping step is more commonly used among the researches since it involves two practical sets, for instance, software components and hardware components, or, more simply, tasks and tiles. C. Marcon [Mar05] defines it as source set of objects $\mathbf{A} = \{\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_p\}$ and a destination set of resources $\mathbf{\Pi} = \{\pi_1, \pi_2, \pi_3, \dots, \pi_q\}$. A mapping is a non-surjective function $f : A \rightarrow \Pi$ that makes a one-to-one association between elements from set A and Π , given that $|A| \leq |\Pi|$.

The complexity to map objects to resources is bigger than the complexity to partition tasks to blocks. It is like that because the complexity to map objects to resources is $O(p!)$, given that p is the number of objects. Consequently, when we combine both techniques, the complexity $O(\text{Bell}(n) \times p!)$ is obtained. For instance, the resulting map of sets \mathbf{A} and $\mathbf{\Pi}$ that has $p = q = 3$ could be one of the following:

$$\{\{\alpha_1\}, \{\alpha_2\}, \{\alpha_3\}\}$$

$$\{\{\alpha_1\}, \{\alpha_3\}, \{\alpha_2\}\}$$

$$\{\{\alpha_2\}, \{\alpha_1\}, \{\alpha_3\}\}$$

$$\{\{\alpha_2\}, \{\alpha_3\}, \{\alpha_1\}\}$$

$$\{\{\alpha_3\}, \{\alpha_1\}, \{\alpha_2\}\}$$

$$\{\{\alpha_3\}, \{\alpha_2\}, \{\alpha_1\}\}$$

2.1.3 Combination of Techniques

In the literature, the step of placing tasks onto processing elements is done by performing partition and mapping steps in a row, or it can be done using exclusively the mapping step. In Section 3 this distinction is presented.

When both techniques are merged, the output of partitioning step is simply applied as input to the mapping step. This approach has a main advantage that it splits the problem into smaller segments. But, essentially, if both approaches (partitioning and mapping or only mapping) are performed exhaustively, the quality of the results is the same, in spite of the fact that they may differ.

2.2 Communication Structures

There are many different topologies of communication structures. Many of them vary slightly from each other. In this section some of the most important and popular architectures are presented.

2.2.1 Point-to-Point

The point-to-point approach is being used as a general solution for SoC communication since the 90s, when the SoC concept arose. It uses dedicated links to communicate between processing elements.

With respect to performance, this topology is considered to be the most efficient since each pair of communication nodes have a dedicated channel. Although it is a good characteristic in terms of performance, it results in a growth of complexity and area. It is more complex because of the fact that designers usually have to design manually the channels between communicating elements and its area grows too much because of the number of data links between communicating nodes. If two nodes have to transfer data in both directions, then the data links must be twice as big as the data links needed by a single direction flow of communication.

Actually, dedicated wires are required for each communication flow, i.e., in some cases, two communicating nodes require two set of linking wires between them, one for sending data from the former node to the second node and another set for the opposite flow. For instance, the number of bidirectional channels to connect all processing elements in an architecture with n processing elements is $n \times n - 1$.

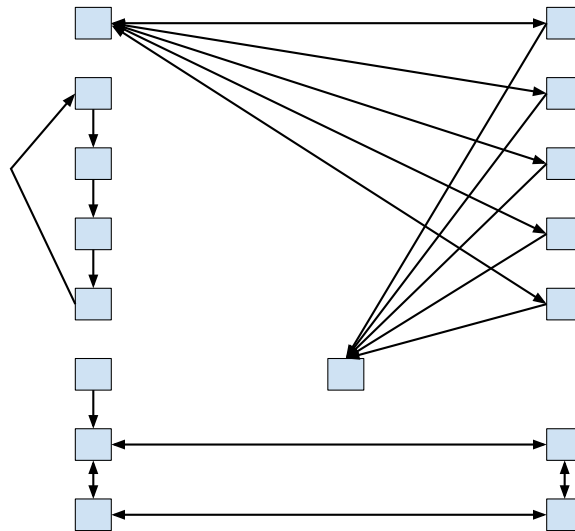


Figure 2.2: Point-to-point infrastructure with sixteen processing elements. See that not every link direction is implemented.

Figure 2.2 shows a point-to-point architecture with sixteen processing elements. Note that not all channels are mandatory, actually, non-communicating processing elements do not need to have channels to interconnect them and there are channels with a single direction.

2.2.2 Bus

A bus is a set of wires that are shared between devices to connect them so they can transmit data to each other. Buses have important characteristics like the smaller complexity of implementation compared to point-to-point and also the efficient use of wires, given that less wires are needed to connect processing elements.

Devices of a bus can be classified as masters or slaves. A common example of master and slave devices are a processor and a memory respectively. The master can control the bus by performing requests to read or write into the bus. On the other hand, the slave can only respond to requests. A traditional graphical representation of a bus is shown in Figure 2.3. Blue squares are used to represent processing elements and the single orange rectangle represents the bus.

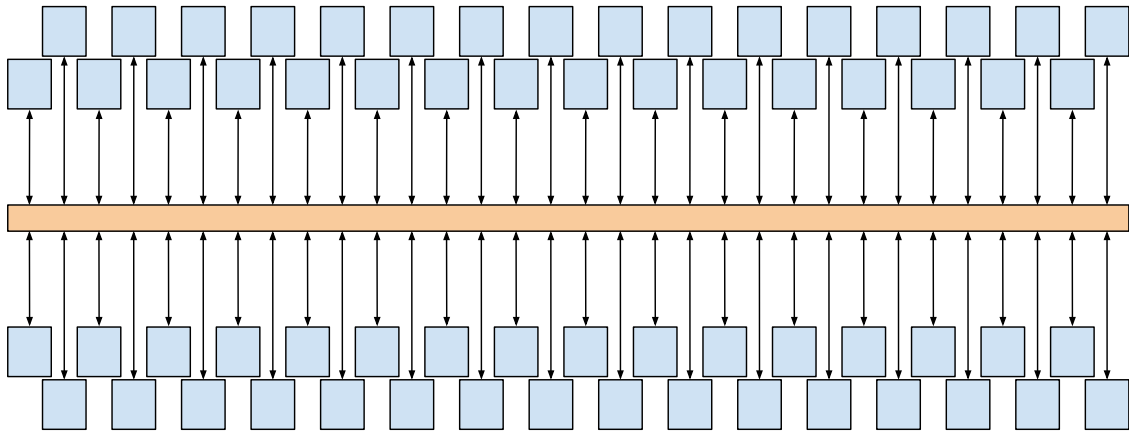


Figure 2.3: Bus connecting 64 processing elements.

A common concept involved in buses implementation is arbitration. Arbitration is responsible for granting access to the bus following an access policy. Such policy is necessary to share the bus and also to avoid data corruption and starvation¹ between bus members.

The most used arbitration policies are round-robin arbitration and arbitration based on priorities. The former treats requests with the same priority and grant access based on a fair sequence. The second one respect a priority list to grant access to the bus. Still related to arbitration, there are more specific techniques like burst mode data transmission [LRD01], where the master negotiates with the arbiter to send multiple words of data over the bus without incurring the overhead of handshaking for each word.

Considering that buses share wires between several processing elements it is not possible to have multiples transmissions simultaneously. Actually, to reach such feature more elaborated techniques like hierarchical buses are needed.

The fact that it is not possible to have multiples transmissions simultaneously results in a very important drawback to buses. As more and more devices are attached to the bus, it becomes a bottleneck of the system. It happens because of the existence of contention to receive access to the media. Contention occurs when one or more elements are requesting access to the media but another element is transmitting through the media.

The fact that contention occurs when a bigger number of processing elements are attached to the media implies in low scalability. Scalability is the ability of systems to grow and handle the growth of work in such a way that there is no degradation of the requisites of the system. Nowadays this characteristics is becoming more important, that is why other network structures like Network-on-Chips became so relevant.

¹Starvation occurs when a device requests access to a resource but the access is never granted though there is no failure or error in the system.

2.2.3 Network-on-Chip

According to [BM06], there are two widely used perceptions of NoC. The former says that NoC is a subset of SoC, and second one that NoC is an extension of SoC. In the first perspective NoCs is an alternative to the communication problem of intra-chip system and is basically referred as an physical media. On the other hand, the second perspective is more broadly concept and also encompass related issues that were noticed with the appearance of NoCs.

As said in Chapter 1, NoCs can be defined as intra-chip communication subsystem, usually composed by a set of routers interconnected by point-to-point communication channels, implementing a chosen topology. One common example of topology used is the Mesh due to its regularity. Figure 2.4 shows a mesh-based network with 64 processing elements - 8 columns and 8 rows of elements. Routers are represented by red squares, while processing elements are represented by blue squares. Arrows indicates links between adjacent components.

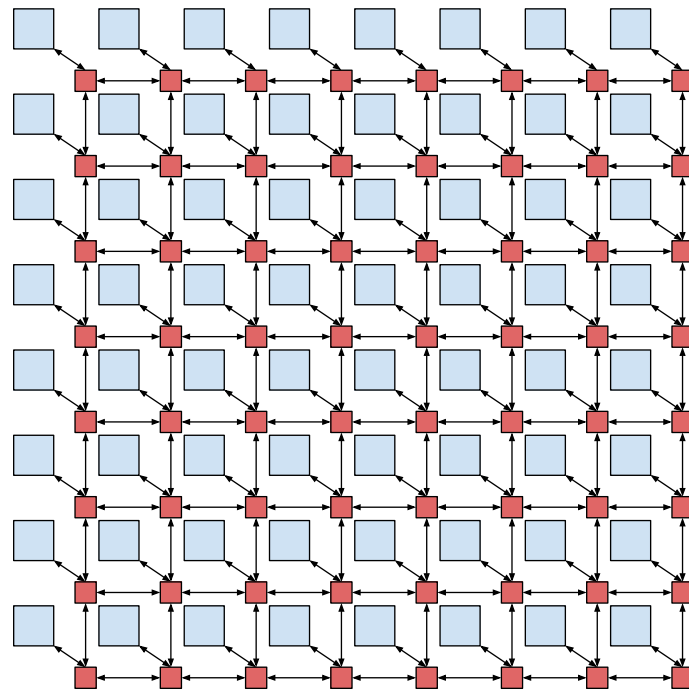


Figure 2.4: 8x8 mesh-based NoC interconnecting 64 processing elements.

In addition to mesh, other well known topologies are torus, tree, ad-hoc organizations and mixed topologies. For instance, torus and mesh are very resembling topologies but their distinctions lies on the fact that nodes in opposite sides in the mesh would be linked in the torus. It means that every router is connected to four adjacent neighbour routers, while in the mesh there are some routers that have only two or three links between neighbour routers. Torus has the advantage of decreasing the average hops needed to transmit data between nodes, but it has the drawbacks that it uses longer wires and routing logic is more significant, thus, it increases hardware complexity.

The arbitration problem, common to buses, also exists in the NoC but it has several differences. The arbiter, instead of arbitrate the access of elements to the shared wires, it is responsible for orchestrate how and the order in that adjacent routers forwards data through links. So, instead of a centralized point of arbitration, NoC distributes this problem between neighbour routers.

There are also new issues that were known in the broad sense of communication but were not previously seen in the field of intra-chip system. For example, concepts like circuit or packet switching, routing algorithms and connection-oriented and connectionless also need to be discussed by system designers. All these concepts were known issues in the broad research field of communication but are now also requiring designers attention in the field of intra-chip systems. Therefore, it is possible to say that in spite of the advantages of network-on-chips, this approach introduces complexity to the design of such systems. More details about these concepts are discussed in the following sections of volume.

Usually, network-on-chips are divided into three main components. Network interfaces, also known as network adapters, that implement the interface by which processing elements are connected to the rest of the network. Their function is to decouple computation from communication. Another component are the routers. They are in charge of routing data across the network. Such components can be implemented in many ways but they use to contain arbiters, buffers, crossbars and a hard-wired algorithm called routing algorithm. And, at last, links composed by channels are in charge of connecting routers to processing elements.

According to [GG00], NoCs have few advantages and disadvantages when compared to buses. Few of the main pros are:

1. Only point-to-point one-way wires are used, for all network sizes. It happens because each channel is composed by two links: one for input and another for output, thus, both of them can be used to forward different packets without interfere each other.
2. The same router may be instantiated again, for all network sizes, what represents reusability to designs using NoCs.
3. Aggregated bandwidth scales with the network size, in opposite to bus.
4. The routing decision can be distributed, what makes fault-tolerant solutions possible. For instance, adaptive routing policies can handle spoiled tiles avoiding to forward packets through them.

The four advantages covered above represent respectively four known concepts to SoC design: *a)* parallelism; *b)* reusability; *c)* scalability; and *d)* reliability. These concepts add value to systems but there are also several drawbacks.

An important drawback is mentioned by authors in [GG00]. They say that NoCs are more complex structures and demand the development of complex high-level drivers. Additionally, decisions about mapping also influence on the performance of systems that adopted NoC. See that

depending on where communicating tasks are placed, more or less hops are necessary to transmit messages. It reflects on other systems constraint like energy, heat and communication throughput. From the perspective of area, intra-chip networks are bigger because there are waiting FIFOs (buffer registers) and logic control segments for arbitration, crossbar and flit forwarding. Such registers and control segments would not exist or would at least be less complex in simpler architectures like buses. Finally, the adoption of network-on-chips requires a re-education of designers and developers in order to understand and better exploit the architecture's potential.

2.2.4 Clustered NoC

A clustered NoC is a special type of network that, instead of linking routers to processing elements, it links routers with a second topology. Some may say that clustered NoCs are multilevel architectures.

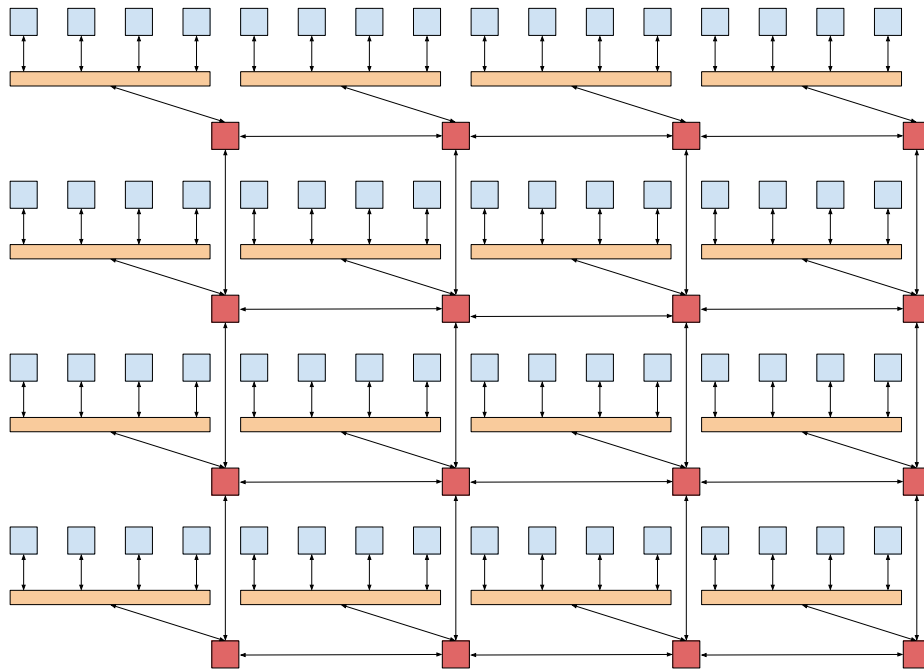


Figure 2.5: 4x4x4 mesh-based NoC.

Obviously there is a huge amount of combinations of different components that can be coupled to compose new architectures and still be labeled as clustered-NoCs, but in this work we assume that cluster-based architectures are composed in a first level by routers and in a second level by buses. In addition, every cluster has the same length of processing elements in spite of the fact that some of these cluster positions may not be used. Two arguments that emphasize the use of such cluster-based architectures according to [TMT12], are their flexibility in terms of form factor which defines the NoC topology and the number of processing elements per cluster and, second, their intrinsic partitioning, that enables the exploration of domain specific application and addressing.

Because of the fact that cluster-based architectures are hybrid solutions of bus and NoC, it is clear to assert that it is a more complex structure than NoC and bus separately. This characteristic obviously demands more attention and effort from designers. Besides, designers have to avoid clusters with too many communicating nodes, since it may cause the same contention that exists in bus-based topologies.

So, this architecture was designed in such a way that it takes benefits from both topologies, the low-cost usage presented in buses coupled with the NoC's greater communication capability and flexibility. It gives to the designer a greater possibility range to map the system elements over the architecture. Consequently, the new mapping possibilities open new design choices that enrich the design space exploration.

An example of cluster-based is shown in Figure 2.5. This example shows a clustered NoC whose routers are used in a first level and clusters are implemented in a second level by buses. In this case, there are 64 processing elements divided by sixteen clusters implemented by buses and interconnected by a mesh-based network-on-chip. The mesh-based NoC has 4 rows by 4 columns and is connected to the buses by their local ports. Routers are represented by red squares, while buses and processing elements are represented by orange rectangles and blue squares respectively. Arrows indicate links between adjacent components.

3. RELATED WORKS

This section is divided in three fields. The first one collects well-known approaches to mapping applications onto Network-on-Chips. Then, a more specific study is presented where only works related to mapping application onto cluster-based MPSoCs are shown. And then, a comparison of this work with the ones previously presented is made.

3.1 Mapping techniques for Network-on-Chip

There are many works that emphasize the importance and complexity of mapping and partitioning application tasks in NoCs. They started as soon as NoCs have arisen as a prominent alternative to design MPSoCs.

One of the most known works that explores the mapping of applications to NoCs architecture was made by Marculescu [HM03]. It captures the number of bit transitions based on an application characterization model, which is described by Marcon [Mar05] as communication-weighted model (CWM). CWM uses the communication-weighted graph (CWG), a simple graph that exposes average communication loads between tasks. In spite of being a simple model, it has few disadvantages, like it cannot represent dependency and order between communications. In addition, it demands efforts from designer to elaborate and label communication arrows with their loads. If arrows are not properly labelled, result precision is decreased from the mapping result.

In [MDM04], the authors present a heuristic algorithm to solve the mapping problem. The algorithm specifically explores bandwidth constraints of mesh NoCs. The objective is to minimize the average delay and it is validated by cycle-accurate simulation of a DSP design modelled in SystemC to which NoC components are added from the `xpipes` library. Their communication model is similar to the model shown in [HM03] and the heuristic algorithm has a behaviour divided into three steps: *i*) initial mapping generation, based on a maximum communication ordering of modules, *ii*) shortest path computation using Dijkstra's algorithm and *iii*) iterative improvement, by swapping pairs of modules and re-computing shortest paths. Another very simple explanation about the approach, is that it labels links with its utilization/load and it explores mappings whose paths between communicating nodes have smaller link loads. One interesting fact of that work, is that one of the activities listed as future work, is to extend the model to map cores onto various NoC topologies, enabling the NoC topology selection.

In [LK03] a two-step genetic algorithm to map applications onto heterogeneous architecture is shown. In the first step a partitioning of applications is presented where each group of tasks were chosen to execute in the same processor type. In the second step occurs the mapping phase. The algorithm was divided to decrease the complexity of partitioning and mapping caused by the explosion of combinations. The approach really decreases it but injecting possible errors, since the initial step estimates processing delays without considering completely the communications delay.

In [MMCM08], the authors collect results of a set of mapping algorithms to map applications to NoCs. The types of algorithms explored comprehend approaches like exhaustive search, stochastic search methods, heuristics, and combinations of stochastic and heuristic algorithms. As input, a communication-weighted graph served as basis to produce a set of experiments representing communication patterns for various applications. The objective of that work is to explore low energy consumption mappings.

3.2 Mapping techniques for cluster-based MPSoCs

In this section, several works that emphasize the mapping of applications onto cluster-based MPSoCs are presented. At the end, Table 3.1 shows in a tabular manner their differences.

The authors from [ANPV10] contributed with a system that provides a multi-bus execution environment where each processor is connected to a bus and the bus-based subsystems communicate via routers connected in a mesh-style configuration. They make a study to evaluate memory access patterns and this study indicates that while a hybrid architecture is preferable, the optimal number of processors on each bus subsystem varies based on the application. This number appears to vary between 1 and 8 depending on the communication requirements of the application. Therefore, they proposed having a reconfigurable interconnect, where the number of cores per bus is variable and assigned by the operating system at run time.

In [TLP⁺10], a bus-mesh hybrid architecture to provide a low latency communication environment for SoC Design was proposed. Their basic idea is to utilize the communication feature of each IP to decide the IP location. In the proposed hybrid architecture, the IPs with heavy traffic and communication affinity are placed in the same bus-based subsystem to avoid hot-spots and reduce the transmission latency. Since the hybrid architecture is based on NoC concept, the router of the hybrid system not only connects with its neighbour routers but also connects to a single IP or a bus-based subsystem. It is noteworthy that a new interface is not needed for each IP in subsystem, and it can further reduce the design cost and power consumption. It also contributes with partitioning and mapping greedy algorithms that has as input the network dimensions and a communication graph.

In [MSA12] a work that explores reconfigurability was also presented. The authors proposed an architecture whose nodes are grouped into some clusters interconnected by a reconfigurable communication infrastructure. From the traffic management perspective, this structure benefits from the interesting characteristics of the mesh topology (efficient handling of local traffic where each node communicates with its neighbours), while avoids its drawbacks (the lack of short paths between remotely located nodes). The work uses few approaches presented by [MDM04] but proposed few adjusts, so that, the existing algorithms could be used with the proposed architecture organization.

In [LSS⁺08], the authors proposed a hierarchical cluster-based customization method. It's methodology uses three separated steps: the first one to do the partition of communicating nodes,

another one for hierarchically compose and organize the network infrastructure and a third one to remove unused network links. This approach is very application specific and its partitioning algorithm has as input a communication-weighted graph, what opens possibilities to improvements with a communication graph that brings more characteristics about communications.

In [TMT12], an indicator based on average Rent's Rule [CS00] was proposed to map applications to cluster-based MPSoCs. The objective is to predict the best form factor of MPSoCs to obtain the maximum system frequency for a given multi-FPGA prototyping platform and a given number of processing elements. To accomplish that, a generic cluster-based MPSoC, based on the Xilinx Microblaze general purpose soft processor, was specifically designed. They build the platform in a manner that the number of clusters and the number of processing elements per cluster is generic. Another characteristic of it, is that both the cluster and the processing elements are all homogeneous. At the end, the target prototyping platform was performed using a six Virtex-5 FPGA board. In a more simple manner, they parametrize their network in three variables: X for NoC width, Y for NoC height and Z for the number of processing elements attached to each NoC tile. Then, they studied the influence of the form factor (X, Y and Z) in the system frequency when prototyping and based on this study they choose the best form factor.

Table 3.1: Comparison of cluster-based works.

Work	Type	Focus	Cluster	Validation	Uniformity	Algorithm
RAMS [ANPV10]	static	low-latency	in-house bus 1-8 processors	PARSEC, SIMICS, NS2	homogeneous	Greedy
[TLP ⁺ 10]	static	low-latency	AMBA	MPEG4, VOPD, ModelSim	homogeneous	SWG and Greedy
[MSA12]	dynamic	energy consumption	mesh	GSM, H263 and VOPD. Xmulator	-	CWG and NMAP
[LSS ⁺ 08]	static	power-saving	hierarchical	MPEG4, VOPD, MWD	homogeneous	Greedy
[TMT12]	static	system frequency	Virtex-5, Fast Simplex Link	-	homogeneous	-
<i>XYC</i>	<i>static</i>	<i>area, latency and number of IPs</i>	<i>HC-MPSoC [Mag13]</i>	<i>MPEG4, VOPD and synthetic apps</i>	<i>homogeneous</i>	<i>MCWM and Simulated Annealing</i>

3.3 Distinction of Proposed Work

All the works presented in this section are related to cluster-based MPSoCs. They also have components known as routers that are in charge of composing the global communication infrastructure. However, our work has a strong distinction from the works [LSS⁺08] and [MSA12] because they do not use buses to implement their clusters, instead, they adopt hierarchical and mesh-based approaches, respectively.

The work being presented here does have affinity with the works [ANPV10] and [TLP⁺10], but, these works apply a more restrictive approach. Both of them use greedy algorithm that have as input a communication weighted-graph introduced by [HM03]. Greedy algorithms are very quick,

but they may fall in very inefficient mappings when the set of inputs is not well defined. In addition to that, the CWG is a graph that characterizes communications as arrows, and each one is labelled with a load, that exposes the amount of data being transferred. Some may consider this approach too simple for the complex systems that are being designed nowadays. Therefore, a map tool that considers more variables is presented. In this map tool different architectures like bus, NoC and clustered-NoC are implemented and a comparison of their performance related to minimum latency is shown.

So, a mapping algorithm that uses a different approach and a communication graph that describes more details about its contents were chosen. The approach adopted is the Simulated Annealing and the communication graph is presented in Section 4. The distinction of this work with the previously presented can be seen at the last line of Table 3.1.

In addition to the mapping algorithm focused on cluster-based architectures, this work aims to provide several tools to help designers to explore design space. For instance, these tools can be used to build simulation set-ups and prototypes. These tools will also be exposed and explained.

4. PROPOSED WORK

The proposed flow is divided into three main logic steps. These steps work with models at different levels of data abstraction, but based on well defined structures it is possible to adapt the data from one step to another automatically. These steps are:

Simulation

Simulation is one of the most important steps since it generates a very sensitive model that will be the basis for the partition step. User will input its tasks and based on a simulation, each message transferred will be recorded. The record of each message will be used to compose the communication model.

Mapping

Optimization algorithms are applied to specific communication task graphs and architectures. The objective is to obtain mapping alternatives to delegate tasks to PEs. These candidates are constructed based on iterative methods that explore the solution space based on a cost function specified by the designer.

End-Source Generation and Target Simulation

With the support of source-code adaptors, simulators and prototype builders, the designer has tools to validate the obtained mapping candidate.

An overview of the whole flow is presented in Figure 4.1. The arrows and the sheets represent respectively dependency of stages and artefacts. These artefacts can be both input and output to other steps. The content syntax of each artefact is explicit in the brackets. Circles are visual representation of automated tools of the proposed flow.

The implemented work can be divided into three groups of software. The former group is responsible for the mapping step. All the models and decisions related to that subject are explained in Section 4.1. The second group of software is related to the Hellfire System. Actually it extends or modifies the Hellfire System in specific points and the work done in this field is presented in Section 4.2. The last group of the tools is a set of independent programs that implements the work flow proposed as the novelty of this work (circles of the Figure 4.1). These programs compose the XYC Toolchain and are presented in Section 4.3.

The name XYC was chosen with the intention to emphasize the ability to model clustered mesh-based architectures with the form-factor $X \times Y \times C$. The variables represent NoC width, NoC height and cluster depth respectively.

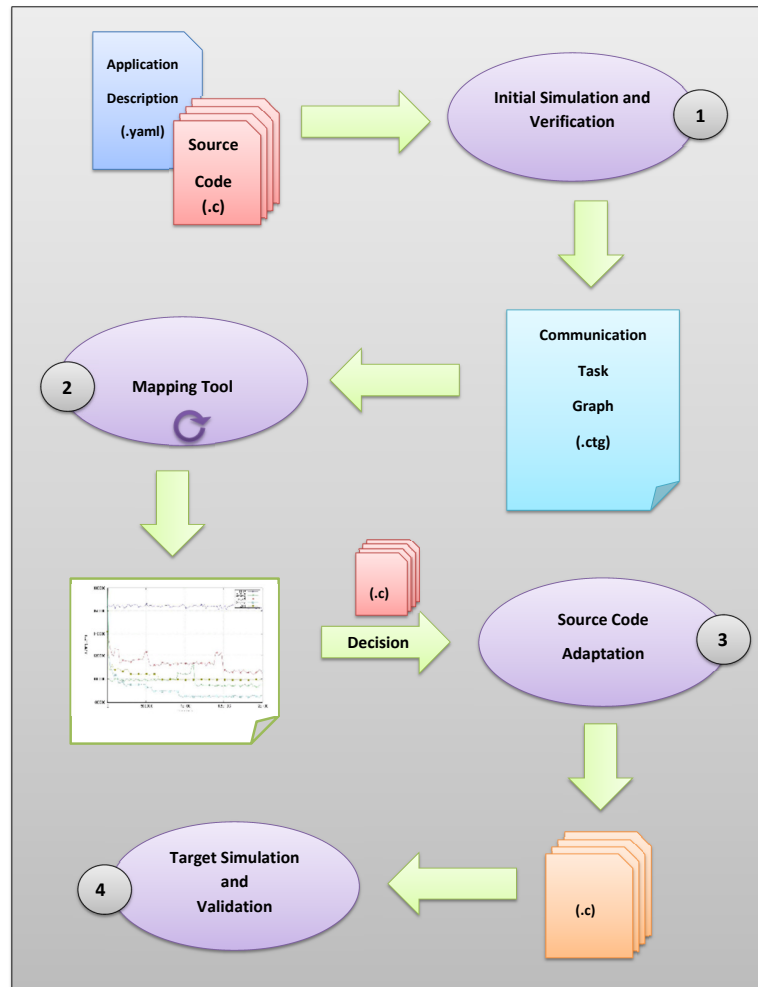


Figure 4.1: Flow to map applications to target architectures.

4.1 Adopted Models and the Mapping Step

There are several parts and decisions needed to implement the mapping step of the work flow. For instance, it is necessary to specify the input, and chose the target architectures, so, these activities are explained in Sections 4.1.1 and 4.1.2. Additionally, algorithms and function costs are needed, so, Section 4.1.3 exposes the function cost adopted while Section 4.1.4 shows the implemented mapping algorithms.

4.1.1 Communication Task Graphs

This section presents the communication-weighted model [Mar05] (CWM), a model that captures the amount of bits transmitted between tasks and its extension. This extension (MCWM) is a superset of the CWM and it is used to increase the precision of mapping costs.

Definition 1: A communication-weighted graph is a directed graph, $G(\Gamma, E)$, where each vertex $s \in \Gamma$ represents a task implemented to a defined processor and labelled with a cpu utilization $0 < u_s \leq 1$. The directed edge $s \rightarrow d (d \in \Gamma)$ denotes the communication flow from task s to task d . Each edge $s \rightarrow d$ is labelled with v_{sd} , the average volume of communication between s and d is represented in bits.

Definition 2: The modified communication-weighted graph is a directed graph, $G(\Gamma, E)$, where each vertex $s \in \Gamma$ represents a task implemented to a defined processor and labelled with a cpu utilization $0 < u_s \leq 1$. The directed edge $s \rightarrow d (d \in \Gamma)$ denotes the communication flow from task s to task d . Each edge $s \rightarrow d$ has two attributes, denoted by v_{sd} and f_{sd} , where v_{sd} is the average volume of communication between s and d , while $0 < f_{sd} \leq 1$ means the average frequency of packets designated to d and injected by s into the communication structure.

CWM takes only into consideration the amount of transmitted data and it does not consider the channel occupation. On the other hand, MCWM tries to explore this characteristic with the f attribute assigned to every communication edge. The choice of a good value for f must be carefully done, once it will play an important role on the results of the experiments and might vary a lot according to different task mappings. As previously mentioned, MCWM is a superset of CWM. So, if every f attribute of the MCWM is assigned with the value 1, then the resulting mapping cost is the same to the equivalent CWM. So, when this information is available, the designer can obtain better estimations, if not, it is still possible to use it according to the widely adopted model.

4.1.2 Abstract Architectures

Based on three basic components it is possible to configure many architectures and compare them using the same metric. These three components are *Routers*, *Processors* and *Channels*. The implemented architectures are *Bus*, *NoC* and *cluster-based NoC*. See that these architectures are high-level representations and are not related to resembling components implemented in Section 4.2.1.

Figure 4.2a shows an architecture composed of 16 processors connected to a bus. All processors share the same channel and have the same access priority to it. Figure 4.2b shows a 4×4 NoC architecture with the same width and height. Processors and router are connected via bidirectional connections each one composed of two physical channels. Figure 4.2c presents a cluster-based NoC. The NoC width is 2, NoC height is 2 and bus length is 4. Routers are connected to each other via connections with two channels. In addition to that, each router is also connected to a cluster of processors via a single channel.

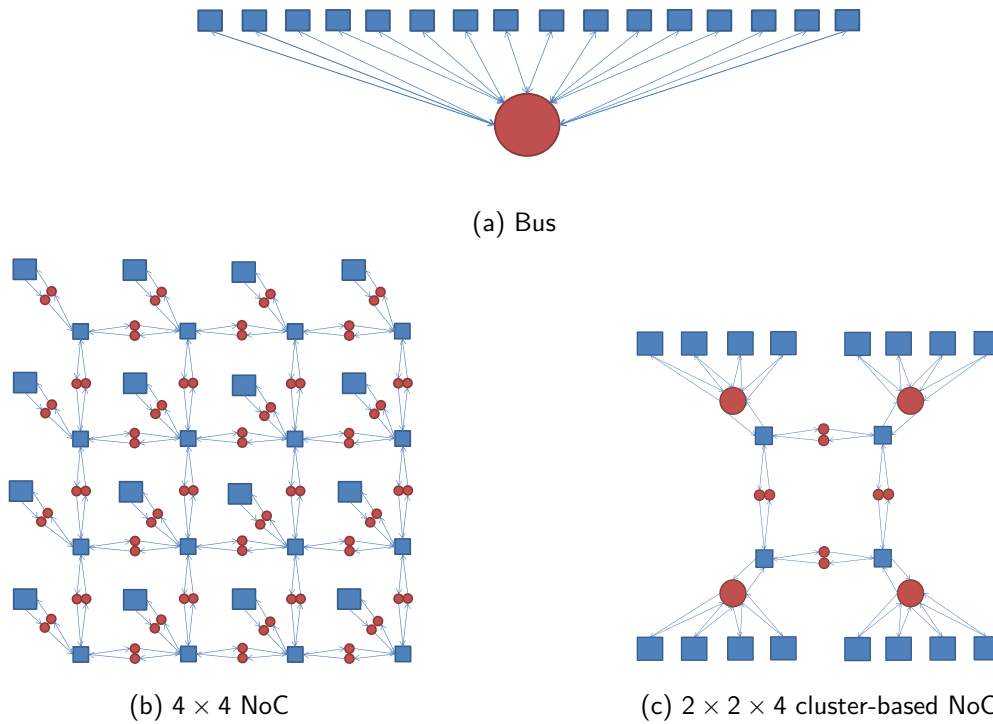


Figure 4.2: Different topologies composed by Routers, Processors and Channels.

4.1.3 Function Cost

Depending on the cost function that designers use, different aspects of the application can be evaluated. Most of the aspects evaluated taken into consideration are energy, heat dissipation or latency. Each of them can be better evaluated depending on the available data informed because the efficiency of each cost function depends both on the characterization of the communication model as well as on the target architecture. The cost function adopted in this work focuses on latency and it is based on the models described in previous sections. Its implementation is described in Figure 4.3.

The *arch* parameter is a high-level representation of the architecture that considers that tasks are already mapped to specific processors of the architecture. The tasks are mapped to processors according to a scheduling policy. Three policies have been implemented in this work: (1) best effort, (2) rate monotonic and a (3) hybrid one. The *arch* parameter has the method *GetComms()* that returns a set of communications edges (*comms*) of the communication graph. The *arch* parameter has also a method that given a specific communication edge, it returns a set of *channels* that a message has to pass before being delivered to its final destination. The method is called *GetPath()* and the set of *channels* returned is called *path*. The *GetPath()* has to implement a routing algorithm to know in advance the channels involved during the transmission process of a given message. In this work the *XY* routing algorithm was chosen because it is a simple non-adaptive routing technique, however our approach is not bounded to this algorithm and more sophisticated algorithms can also be used. Finally, just as the parameter *arch* does, *channels* have also a method

```

1: function Cost(arch)
2: cost  $\leftarrow$  0
3: comms  $\leftarrow$  arch.GetComms ()
4: for all comm  $\in$  comms do
5:   path  $\leftarrow$  arch.GetPath (comm)
6:   for all channel  $\in$  path do
7:     channel_comms  $\leftarrow$  channel.GetComms ()
8:     for all other  $\in$  channel_comms do
9:       if other  $\neq$  comm then
10:        cost  $\leftarrow$  cost + other.v * other.f
11:       else
12:        cost  $\leftarrow$  cost + comm.v
13:       end if
14:     end for
15:   end for
16: end for
17: return cost

```

Figure 4.3: Description of the cost function.

called *GetComms()*, but in that case, the method returns all *comms* that pass through the specific *channel* before being delivered to the message destination. With this information it is possible to obtain the costs to send messages in every single *channel* and sum these costs to obtain an estimation of *arch*'s mapping quality.

The meaning of the returned *cost* represents the number of bits (*in the worst case*) necessary to be forwarded before a message can be delivered. Considering that the objective function is implementing a worst case approach, the absolute value returned should not be taken as a meaningful value. However, when using the same metric with other architecture models, it is possible to use the cost variation to evaluate the architectures and consequently take a decision based on that difference. In other terms, this technique calculates the mapping cost and bigger values mean that more obstructed are the channels involved in the transmission of the messages. This technique is also known as *PathLoad* exploration.

4.1.4 Mapping Algorithms

The Mapping step has few requirements to be properly executed. It requires a communication task graph, which is obtained at the end of the Initial Simulation step. It requires a high level architecture model to be the target topology of the applications. It also requires a function cost to evaluate a specific characteristic of the system. These requirements were already explained in the previous section, but there is still another necessary feature to realize the mapping step: a mapping algorithm. The mapping algorithms basically try different mapping candidates against each other to obtain a convenient mapping solution. Four mapping algorithms were implemented until now, but

it is possible to develop other algorithms. These implementations are based on two basic algorithms and their variations. The algorithms are Random Algorithm and Simulated Annealing, and their characteristics are detailed below.

Random Algorithm

The random approach is a primitive algorithm that is not usually adopted since it is possible to obtain better results with other techniques. However, it serves as basis for many optimization algorithms, including Simulated Annealing. Even though this technique is rarely adopted, the algorithm was implemented in order to obtain a baseline for comparisons. The algorithm is illustrated in Figure 4.4.

```

1: function Random(arch)
2: best ← RandomMapping(arch)
3: best_cost ← Cost(best)
4: while stop_condition do
5:   alternative ← RandomMapping(arch)
6:   alternative_cost ← Cost(alternative)
7:   if alternative_cost ≤ best_cost then
8:     best ← alternative
9:     best_cost ← alternative_cost
10:  end if
11: end while

```

Figure 4.4: Random mapping algorithm.

Simulated Annealing

Simulated Annealing (SA) is a generic probabilistic meta-algorithm for global optimization problems, namely locating a good approximation to the global optimum of a given function in a large search space [KGV83]. The technique was developed in 1983 and forms the basis of an optimization technique for combinatorial and other problems. It is widely used in a large range of applications and is inspired by the annealing technique in metallurgy. This technique involves heating and controlled cooling a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial position and wander randomly through states of higher energy.

In spite of the fact that Simulated Annealing is a widely adopted algorithm, there are emerging technologies like clustered-NoCs that were not yet addressed by this approach. With that in mind, this work focuses on the study of three implementations of the Simulated Annealing that handles the tasks mapping issue in clustered architectures.

The algorithm, in its basic form, iterates random searches of the space solution, but unlike the random algorithm, it has a mechanism to avoid becoming trapped in a local minimum. In

some iterations, the algorithm may accept changes of the best known solution for another one that increases the cost function (assuming it is a minimization problem). These changes that increase the cost are accepted according to a probability $p = \exp(\Delta c \div T)$, where Δc denotes the increase of the cost function and T is a control variable. At every iteration the variable T (popularly known as temperature) decreases, and consequently, the probability of not accepting bad mappings candidates increases.

The implementation of the Simulated Annealing can be seen in Figure 4.5. There is another detail that differs SA from the random algorithm beyond the mechanism to accept or not a new solution. On line 5, the method `Copy()` copies the whole best known solution. Later, on line 6, it applies a modification using method `Modify()`. This method changes the location of a single task and calculates the mapping costs again. So, unlike the random algorithm which generates a completely different mapping at each iteration, the implementation of the Simulated Annealing only performs a modification in an architecture representation that is a copy of the momentary best mapping.

```

1: function SimulatedAnnealing(arch)
2: best  $\leftarrow$  InitialMapping (arch)
3: best_cost  $\leftarrow$  Cost (best)
4: while stop_condition do
5:   alternative  $\leftarrow$  Copy (best)
6:   alternative  $\leftarrow$  Modify (alternative)
7:   alternative_cost  $\leftarrow$  Cost (alternative)
8:    $\Delta$   $\leftarrow$  best_cost - alternative_cost
9:   if  $\exp(\Delta \div \textit{temperature}) \geq \textit{Random}()$  then
10:    best  $\leftarrow$  alternative
11:    best_cost  $\leftarrow$  alternative_cost
12:   end if
13:   temperature  $\leftarrow$  temperature  $\times$   $\alpha$ 
14: end while

```

Figure 4.5: Traditional implementation of the Simulated Annealing.

Nested Simulated Annealing

There is also another variation of the Simulated Annealing that comprehends two nested loops. The external loop implements a global search technique, the random for instance. And the internal loop performs simple modifications, like changing the position of a single task. Dividing the external mapping algorithm into two nested loops modifies the probability of finding a global minimum instead of a local minimum. This happens because the algorithm explores space solution randomly, since the internal loop searches for a local minimum inside each valley (range of solution candidates from the design space exploration) while the external loop changes randomly the valleys.

```

1: function NestedSimulatedAnnealingV1(arch)
2: best  $\leftarrow$  InitialMapping (arch)
3: best_cost  $\leftarrow$  Cost (best)
4: while outer_stop_condition do
5:   alternative  $\leftarrow$  Copy (best)
6:   alternative  $\leftarrow$  ModifySubstantially (alternative)
7:   while inner_stop_condition do
8:     alternative  $\leftarrow$  ModifySlightly (alternative)
9:     alternative_cost  $\leftarrow$  Cost (alternative)
10:     $\Delta$   $\leftarrow$  best_cost  $-$  alternative_cost
11:    if  $\exp(\Delta \div \textit{temperature}) \geq \textit{Random}()$  then
12:      best  $\leftarrow$  Copy (alternative)
13:      best_cost  $\leftarrow$  alternative_cost
14:    end if
15:    temperature  $\leftarrow$  temperature  $\times$   $\alpha$ 
16:  end while
17: end while

```

Figure 4.6: Second implementation of the Simulated Annealing.

```

1: function NestedSimulatedAnnealingV2(arch)
2: best  $\leftarrow$  InitialMapping (arch)
3: best_cost  $\leftarrow$  Cost (best)
4: while outer_stop_condition do
5:   list  $\leftarrow$  []
6:   while inner_stop_condition do
7:     alternative  $\leftarrow$  Copy (best)
8:     alternative  $\leftarrow$  Modify (alternative)
9:     alternative_cost  $\leftarrow$  Cost (alternative)
10:    list  $\leftarrow$  list + [(alternative, alternative_cost)]
11:    temperature  $\leftarrow$  temperature  $\times$   $\alpha$ 
12:  end while
13:  alternative, alternative_cost  $\leftarrow$  Min (list)
14:   $\Delta$   $\leftarrow$  best_cost  $-$  alternative_cost
15:  if  $\exp(\Delta \div \textit{temperature}) \geq \textit{Random}()$  then
16:    best  $\leftarrow$  Copy (alternative)
17:    best_cost  $\leftarrow$  alternative_cost
18:  end if
19: end while

```

Figure 4.7: Third implementation of the Simulated Annealing.

The implementation of the Nested Simulated Annealing is shown in Figure 4.6. This implementation has two different methods: `ModifySubstantially()` and `ModifySlightly()`. Both methods are called with tasks already mapped to the architecture. The first method changes the position of $\lceil t \div 2 \rceil$ tasks of the architecture, given that t is the total number of tasks. We consider that changing the position of $\lceil t \div 2 \rceil$ tasks is a substantial modification because half of the tasks are moved. The later method does the migration of one task and is equivalent to the method `Modify()` from the previous implementation.

Figure 4.7 describes the implementation of a second variant of the Nested Simulated Annealing. The cost functions calculated in the inner loop and the modification performed by the `Modify()` method are stored in a list sequentially. After each iteration of the outer loop, the best *move* stored on the list is chosen as the new candidate to be tested in the acceptance test. The method `Min()` returns respectively the architecture with a mapping description and its cost. The variable *inner_stop_condition* is a simple controller that iterates $\lceil t \div 2 \rceil$ times.

4.2 Hellfire System

The Hellfire System is a set of subsystems and modules. These modules are basically tools that support designers to build embedded systems. Between the group of main modules we can mention the following:

Hellfire OS

Embedded realtime operating system.

Hardware Modules

Set of modules as processor, bus, NoC and Clustered NoC described in HDL.

MPSim

Simulator that emulates all hardware modules.

Web-Framework

An online Integrated Development Environment.

The most important of the modules probably is the HellfireOS (HFOS). It is a real-time operating system (RTOS) developed intending to ensure maximum flexibility on its configuration and allow a high-level platform customization. In order to allow such features, the HFOS was implemented in a modular way, where each module corresponds to some specific functionality.

The HFOS is organized in layers and all hardware-specific functions are defined in the first layer, known as HAL (Hardware Abstraction Layer). The uKernel lies just above it and the communication, migration, memory management and mutual exclusion drivers, as well as the API are placed over the uKernel layer. The user applications belong to the top layer. Due to its modular

implementation, the HellfireOS is easily portable to others architectures, requiring only the rewrite of hardware-dependent functions, implemented in the HAL.

In order to decrease the kernel final size, allowing HFOS to be used even in architectures with severe memory limitations, parameters such as maximum number of user tasks, stack and heap size and drivers are configurable. The user applications are written using the C programming language and the HellfireOS API.

All the architectures implemented and the drivers are focused to run with or over Hellfire OS. Based on that, the implemented communication medias are presented below and after that an explanation of the communication driver is also given.

4.2.1 Implemented Architectures

Architecture like Bus, NoC and CNoC are already implemented in the Hellfire System but they were described using a hardware description language [AFM⁺10]. Not all architectures were represented in high-level languages in such a way that it is possible to run fast simulations with them. Due to that, high-level implementations of the existing architectures (Bus and NoC) were reviewed and the behaviour of the clustered NoC was implemented in C language.

The implementation of these three architectures have relevant distinctions and they are described below. In addition to that, some components common to the three architectures are also presented.

Common Concepts

There are some concepts that are common and presented in the three studied architectures. They are Network Interfaces, Flits, Packets, Message and Memory Mapped Addresses.

Messages are data sent between communicating tasks. The message before being injected into the communication media is divided into packets. Packets can have fixed or flexible size but in this work fixed-sized packets were chosen.

Packets are composed by smaller parts, the flits. Flits represent the smallest amount of data meaningful for the communication media. Usually flits can have two classification: header flits and payload flits. Header flits are meaningful for the communication media because they specify useful information for the forwarding of the following packet's flits. On the other hand, payload flits are irrelevant data for the communication media, i.e., they only have to be passed on.

The division of the messages into packets is performed by the communication driver. The driver is also responsible for composing packet header and controlling data injection and reception. To interact with the communication media the driver uses memory addressed registers. These registers are input and output ports for the network interface (NI).

NIs are composed by two FIFOs. One of the FIFOs is responsible for storing data injected from attached devices and for forwarding data to the network media. The other FIFO is responsible for the opposite data flow. It stores data from communication media and forwards the data to the attached devices.

Bus

Figure 4.8 shows the block diagram of the bus specification. The bus is composed by parametrizable network interfaces and a logic control that handles arbitration and flits forwarding. The blue squares are the attached devices. In particular, this work allows only processors to be attached to the bus. The main advantage of this architecture is the resulting area that is smaller than the next architectures because it has a single simple module, the own bus.

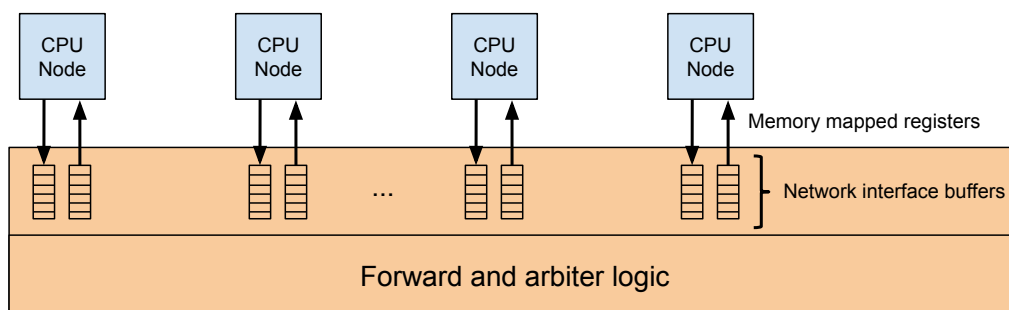


Figure 4.8: Bus specification.

Network-on-Chip

For our NoC implementation we use HERMES NoC [MCM⁺04], which implements a mesh topology and is composed by routers, buffers and controllers of routers information (switch control). HERMES routers can have up to five operating channels. Channels are composed by buffers fed by incoming and outgoing ports. Ports are wires that connect adjacent routers.

As mentioned above, routers can have from two to five channels because routers allocated in the border of mesh do not have all adjacent routers, thus, the respective channels are unnecessary. Moreover, not every router need to be attached to a processing element by the port known as local port. In such situations these channels are unnecessary too.

An overview of the NoC Router is shown in Figure 4.9. Blue squares represent the devices attached to the media while red components are the parts that compose communication media. Please, pay attention the fact that each router has one buffer for each port. It was made like that because outgoing flits go directly to the router adjacent incoming buffer.

The HERMES router enables designers to choose between flits from 8 to 64 bits. However, the Hellfire System adopt a fixed size of 16 bits per flit. In addition to the flit size, HERMES routers can be configured with other options. For instance, it is possible to choose the routing algorithm,

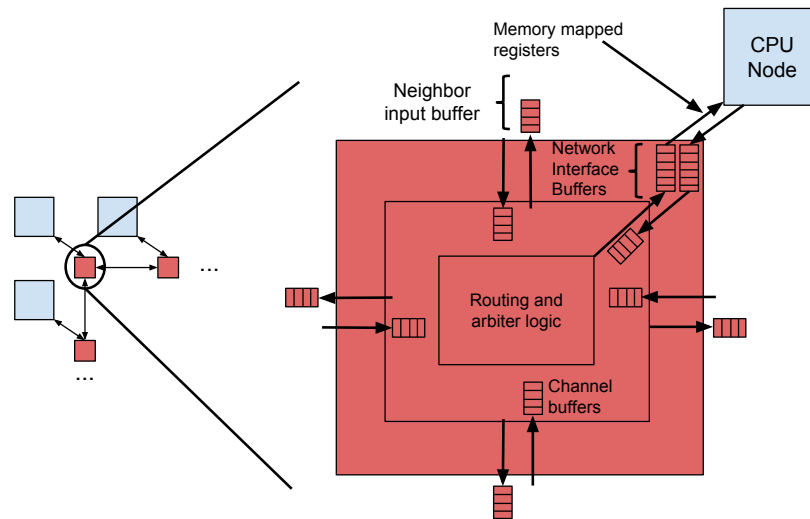


Figure 4.9: Router specification.

the control flow strategy, the buffers' size and the arbiter policy. This work uses the XY routing algorithm since it is a simple and deadlock free algorithm. This work uses rotative arbiter due to the fact that it does not suffer from starvation and also because it's algorithm is not complex. Finally, this work chose handshake as the control flow strategy because in the future the Hellfire Systems will enable heterogeneous mixed architecture and such control strategy is favourable to this reason.

Clustered Network-on-Chip

The implementation of the high-level clustered NoC respects precisely the behaviour of the CNoC implemented by GSE [Mag13]. In addition to that, they use identical routers as the NoC specification.

However, unlike NoC, clustered NoC does not link network interfaces to routers' local port buffers. Instead of that the local port buffers are linked to another module called cluster interface (CI). CI is a module with the same behaviour as network interface but it is an intermediate step to forward packets from bus to NoC routers and vice versa. In Figure 4.10 it is possible to see how bus and NoC router are linked by the cluster interface and the local port buffer.

4.2.2 Communication Driver

The communication protocol was implemented as a Hellfire OS driver. Therefore, it is possible to evaluate different communicating applications using the tools available by the Hellfire Web-Framework.

The communication driver implements a stack of software with four layers. The top layer is the Application Layer and it offers a very simple API to send and receive messages. Table 4.1 shows the primitives available by the operating system to send and receive messages. The layer

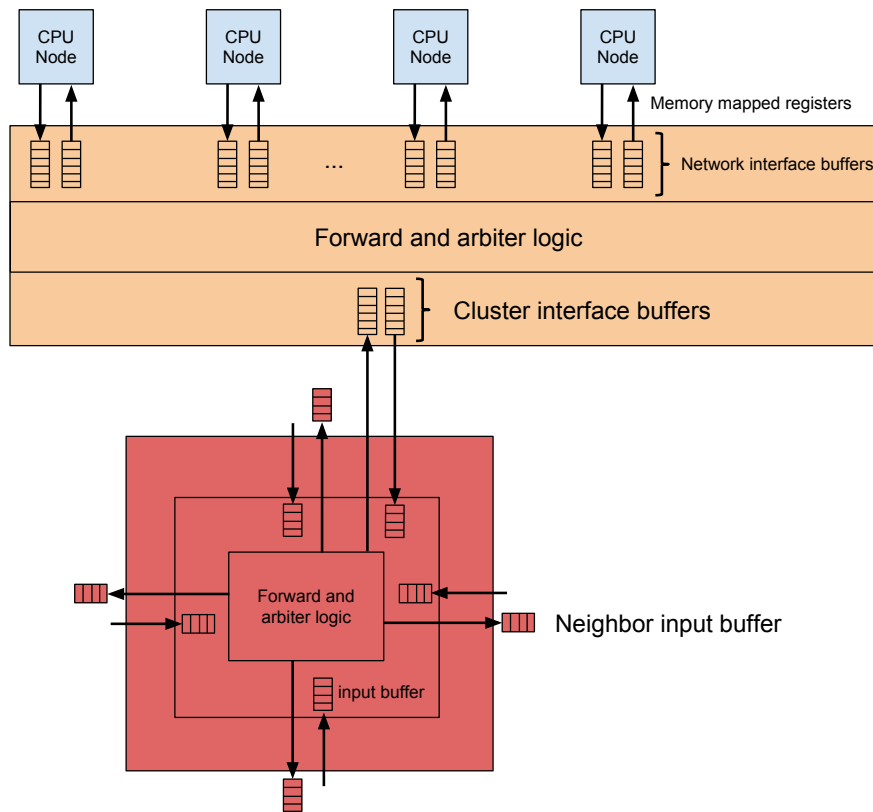


Figure 4.10: CNoC Local Interface Structure.

below application layer is the Transport Layer. This layer basically split messages into packets and reassemble messages. It is also responsible for handling unordered packets or messages whose time-outs were reached. After that, lies the Network Layer. The network layer adds or removes the header required by the communication media. We will see that different media require different headers. Finally, the last layer is the Link Layer. This last layer basically interacts with Networks Interfaces through memory mapped addresses.

Table 4.1: Hellfire communication primitives.

Signature
uint32 HF_NB_Send(uint16_t target_cpu, uint8_t target_id, uint8_t buf[], uint16_t size, uint32_t timeout)
uint32 HF_Send(uint16_t target_cpu, uint8_t target_id, uint8_t buf[], uint16_t size)
uint32 HF_NB_Receive(uint16_t *source_cpu, uint8_t *source_id, uint8_t buf[], uint16_t *size, uint32_t timeout)
uint32 HF_Receive(uint16_t *source_cpu, uint8_t *source_id, uint8_t buf[], uint16_t *size)

The interaction of the processing element with Networks Interfaces is made by three memory mapped addresses, in addition to a interrupt signal. The registers are: NET_STATUS, NET_READ and NET_WRITE. The former is a read register that answers true or false. True means that the network interface buffer that injects flits into the communication media is available. Read operations to this registers should be performed in a dedicated segment code, therefore, they should be performed inside a mutual exclusion controller. The NET_READ register is a read register and it pops flits from the network interface. Reads from this register should be made by the communication interrupt handler when a interrupt provided by the communication media is triggered.

Finally, the NET_WRITE register is used to insert flits to the network interface and should be called after a polling process on the NET_STATUS register.

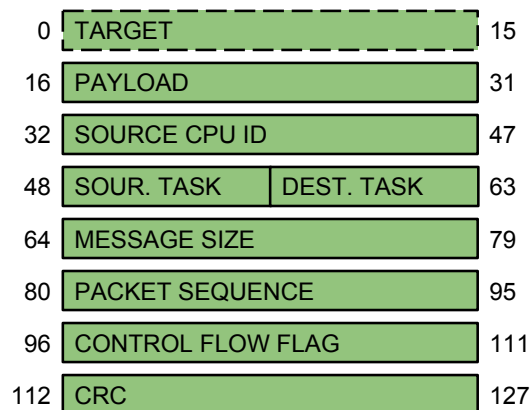


Figure 4.11: Communication protocol header.

Figure 4.11 shows the communication protocol header. Flit target and flit payload compose the header required by the media. Target varies with the media, but payload has the same meaning in every architecture and means how many flits are remaining in the body of a packet.

SOURCE CPU ID is a decimal field that answers what was the processing element that injected the packet into the network. This data is useful for the application layer since it is possible to identify in run-time the sender of the received messages.

SOURCE TASK and DESTINATION TASK are useful for the transport layer. With this data, it is possible to address messages to nodes running more than one task, for instance, nodes running operating systems and their applications. Other flits important for the transport layer are MESSAGE SIZE and PACKET SEQUENCE. They are complementary data and they are used by transport layer on the reassembling message process.

Other optional data of the header are CONTROL FLOW FLAG (CFF) and CRC. CFF is a flag that tells driver if an packet received should generate an acknowledge message. CRC is an abbreviation Cyclic Redundancy Check and is used to check whether any data corruption occurred.

Figure 4.12 shows how the TARGET flit differs from one communication media to another. At that point it is important to remember that although Hellfire System tools enable different sizes for the flit, this work adopts flits with 16 bits as its default flit size. Actually, bigger sizes can be chosen but smaller sizes would not fit with cluster-based architectures.

The bus target flit is the simplest one. It basically uses the eight least significant bits to indicate the target CPU. It means that 256 processing elements can be attached to the bus, though we do not encourage the use of bus with such amount of elements. The remaining most significant bits are unused.

The mesh network-on-chip also considers only the eight least significant bits to address packets to their destinations. COL ID and LINE ID are fields responsible for identifying the position of the destination of the packet. COL ID specifies target cpu on the x-axis while LINE ID specifies

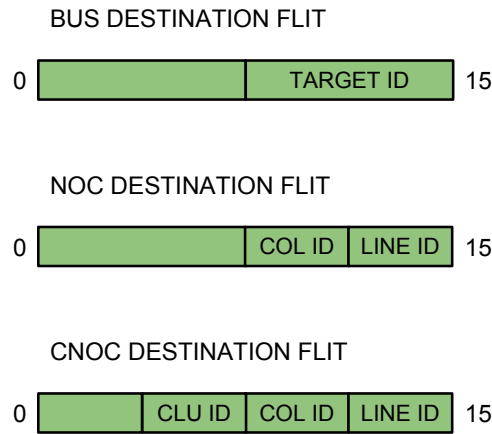


Figure 4.12: Target flit specification.

the y-axis target position. Considering that COL ID and LINE ID are four bits fields, it is possible to build NoCs with 256 tiles (16×16).

As it was already explained, cluster-based NoCs are clusters connected by network routers. Considering that the networks are all implemented using the mesh topology, we say that COL ID and LINE ID identifies target cluster ID on x-axis and y-axis respectively. In addition to these fields, there is still the cluster position ID field. It is useful to identify the CPU position inside each cluster. Considering that the three fields are 4 bits each, it is possible to build CNoC architectures with 256 clusters and each cluster can support up to sixteen processing elements. The remaining most significant bits are unused.

4.3 XYC Toolchain

At that point we present the set of tools that implement the proposed work flow. See Section 4 for more details about that flow. Pay attention to the fact that the output of the each tool serves as input for the next tool on the software chain.

4.3.1 XYCSIM - XYC Simulator

XYCSIM is essentially a platform generator. The platform obtained using the tool can be simulated using OVP [OVP12], a virtual platform simulator, or even using the simulators implemented in GSE. The objective of this simulation is to obtain a MCWG. The MCWG is used as input by the next tool, the XYCMAP.

In technical terms, the XYCSIM is a composition of three scripts. The former script builds a makefile used to compile the platform. The next script creates the OVP platform. This platform is a executable program that makes run-time instantiations of OVP models. The last script is responsible for generating a C header file. This header file contains macros used to identify every

task of the task set. There are two mandatory inputs: `-tasks` and `-procs`. `-tasks` is a string parameter that points to a file that contains a list of tasks. `-procs` is a string parameter too but it points to a file that list processors and describes their characteristics. Figure 4.13 shows tool's interface.

```

oliver@rothaus: ~/mestrado
oliver@rothaus:~/mestrado$ xyccsim -h
usage: build_makefile.py [-h] -tasks TASKS [-procs <PROCSFILE>]

Builds makefile.

optional arguments:
  -h, --help            show this help message and exit
  -tasks TASKS          File that contains the application description.
  -procs <PROCSFILE>   File that contains the processor descriptions.
usage: build_platform.py [-h] -tasks <TASKSFILE> -procs <PROCSFILE>

Builds OVP platform.

optional arguments:
  -h, --help            show this help message and exit
  -tasks <TASKSFILE>   File that contains the application description.
  -procs <PROCSFILE>   File that contains the processor descriptions.
usage: build_header.py [-h] -tasks <INPUTFILE> [-procs <INPUTFILE>]
                        [-number_of_flits <NUMBER_OF_FLITS>]

Builds header file that contains the applications' macros.

optional arguments:
  -h, --help            show this help message and exit
  -tasks <INPUTFILE>   File that contains the application description.
  -procs <INPUTFILE>   File that contains the application description.
  -number_of_flits <NUMBER_OF_FLITS>
                        Variable needed to build header.
oliver@rothaus:~/mestrado$

```

Figure 4.13: XYCSIM help command.

4.3.2 XYCMAP - XYC Mapper

Figure 4.14 shows the text-based user interface of the tool. The figure shows the range of parameters available. The most important parameters are the `-arch`, `-x`, `-y`, `-c`, `-b`, `-input`, `-output`, `-policy`, and `-algorithm`. The `-input` parameter is a name to file that contains the desired communication task graph. This graph in most of the times is the output of the initial simulation of the flow. The `-output` parameter is a file name of the file that will contain the cost of each mapping tool iteration. The parameters `-x`, `-y`, `-c`, and `-b` are needed to describe the target architecture. these parameters are strongly coupled to the `-arch` that indicates the desired target architecture. If the user defines `-arch` as NoC, then the user must fill the parameters `-x` and `-y`. If the user chooses `-arch` as CNoC, then the `-x`, `y` and `-c` are mandatory. On the other hand, if the user defines `-arch` as Bus, only the `-b` parameter is required and the others are ignored.

```

oliver@rothaus: ~/mestrado
oliver@rothaus:~/mestrado$ xycmap -h
usage: xyc.py [-h] -arch {NoC,CNoC,Bus} [-x <X> [<X> ...]] [-y <Y> [<Y> ...]]
[-c <C> [<C> ...]] [-b <B> [<B> ...]] -tasks <TASKSFILE> -comms
<COMMSFILE> -procs <PROCSFILE> -policy {BE,RM}
[-output <OUTPUTFILE>] [-output_map <OUTPUTMAPFILE>]
[-plot <PLOT_FILE>] [-algorithm {Rand,SA,SAN,SAN2}] -n <N>
[-label <LABEL>]

Map a task set to a cluster-based architecture.

optional arguments:
-h, --help                show this help message and exit
-arch {NoC,CNoC,Bus}      target architecture.
-x <X> [<X> ...]          values allowed for x-axis.
-y <Y> [<Y> ...]          values allowed for y-axis.
-c <C> [<C> ...]          values allowed for c-axis.
-b <B> [<B> ...]          values allowed for b-axis.
-tasks <TASKSFILE>        task set input file.
-comms <COMMSFILE>        comm set input file.
-procs <PROCSFILE>        processor set input file.
-policy {BE,RM}           scheduling policy to apply.
-output <OUTPUTFILE>      cost list output file.
-output_map <OUTPUTMAPFILE>
                           mapping output file.
-plot <PLOT_FILE>         file to store the resulting plot.
-algorithm {Rand,SA,SAN,SAN2}
                           Which mapping algorithm should be applied.
-n <N>                    number of iterations to run.
-label <LABEL>            if set, output files are all going to be named in
                           function of label's name. This option overwrites the
                           "-output" option.

oliver@rothaus:~/mestrado$

```

Figure 4.14: XYCMAP help command.

Considering that the parameters $-x$, $-y$, $-c$, and $-b$ are set of lists results in an important effect on the mapping phase. It makes the mapping process aggressively more time consuming because of the growth caused by the combination of acceptable configuration. To mitigate this growth of the explorable space a simple policy was used. Once an arrangement that respects the scheduling policy is found, no new arrangements can use more processing elements than that one. This simple alternative is enough to prevent the algorithm from testing too consuming arrangements, while easily covers simple decisions that designers have to take, like choosing square architectures, or irregular ones, like rectangular or even pipelined ones. Alternatively, the designer can input single values for each axis, then the mapping algorithm is applied to a single architecture configuration. See that this feature is one of the goals of the work, once it explores architectures with smallest number of processing elements.

The $-policy$ parameter defines guard for accepted tasks arrangements. At each iteration of the mapping tool a new task arrangement is obtained. This arrangement is tested against a scheduling policy. If the test does not succeed, the arrangement is discarded. Currently, only Rate Monotonic (RM) and Best Effort (BE) were implemented, but it is possible to implement other policies like single task execution per node or other real-time scheduling algorithms.

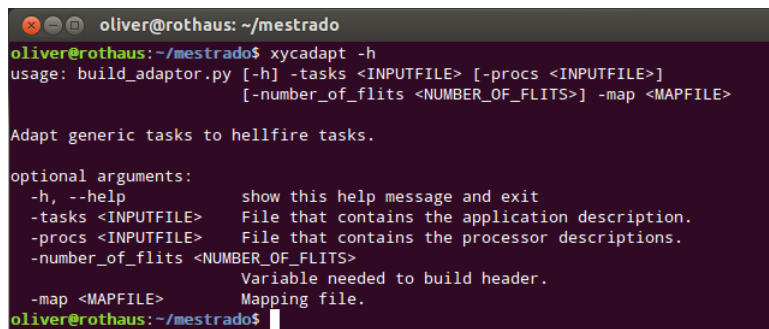
The last important parameter is $-algorithm$. It defines the mapping algorithm that should be used to map tasks to processors. Currently, there are four algorithms implemented but it is possible to add new ones. The ones implemented were presented in the previous sections of this document.

There are also others not so relevant parameters. They do not cause any consequence to the mapping results but are useful for designers to obtain preprocessed output files. These files are easier to parse using scripts to obtain analytical tables and figures.

4.3.3 XYCADAPT - XYC Adaptor

The third tool of the toolchain performs an important task of the proposed flow. It takes ANSI C tasks and converts them to Hellfire OS tasks based on a mapping file. To ease the transformation the ANSI C tasks require some annotations using comments.

The parameters `-tasks` and `-procs` are string parameters that point to two distinct files, one that contains a list of tasks and their attributes, and the other a list of processors and their characteristics. The other parameter that is the most important is the `-map` file. This file contains a mapping arrangement that specifies a network communication infrastructure and also defines the location of each task (from `-tasks` parameter) to a tile of the network. Figure 4.15 shows the tool's interface.



```

oliver@rothaus: ~/mestrado
oliver@rothaus:~/mestrado$ xycadapt -h
usage: build_adaptor.py [-h] -tasks <INPUTFILE> [-procs <INPUTFILE>]
                        [-number_of_flits <NUMBER_OF_FLITS>] -map <MAPFILE>

Adapt generic tasks to hellfire tasks.

optional arguments:
  -h, --help            show this help message and exit
  -tasks <INPUTFILE>   File that contains the application description.
  -procs <INPUTFILE>   File that contains the processor descriptions.
  -number_of_flits <NUMBER_OF_FLITS>
                        Variable needed to build header.
  -map <MAPFILE>       Mapping file.
oliver@rothaus:~/mestrado$

```

Figure 4.15: XYCADAPT help command.

4.3.4 XYCPRO - XYC Prototype builder

Finally, the last tool of the XYC toolchain is responsible for building a runnable instance of the architecture. Actually the target architecture depends directly on the mapping parameters given to the XYCMAP tool. The architectures available are those shown in Section 4.2.1.

Similarly to the XYCSIM tool, the XYCPRO is also a union of three other smaller builders. One tool is responsible for creating a makefile that will be used to compile the platform and operating system binary objects. The second builder creates an OVP platform model. The last builder creates a C header file that identifies tasks. This header file is especially important because it provides information to properly address packets given that packet header varies depending on the chosen architecture. See Section 4.2.2 for more details about it.

Figure 4.16 shows the text-based user interface for the prototype builder. The parameters are identical to the parameters requested by XYCADAPT except for the `-tasks` parameter because at that time the parameter should be fulfilled with an already adapted task description file.


```

oliver@rothaus: ~/mestrado
oliver@rothaus:~/mestrado$ xycpro -h
usage: build_makefile.py [-h] -tasks TASKS [-procs <PROCSFILE>] -map <MAPFILE>
[-number_of_flits <NUMBER_OF_FLITS>]

Builds makefile.

optional arguments:
-h, --help            show this help message and exit
-tasks TASKS          File that contains the application description.
-procs <PROCSFILE>   File that contains the processor descriptions.
-map <MAPFILE>       Mapping file.
-number_of_flits <NUMBER_OF_FLITS>
                        Variable needed to build header.
usage: build_platform.py [-h] -tasks <TASKSFILE> -procs <PROCSFILE> -map
<MAPFILE> [-number_of_flits <NUMBER_OF_FLITS>]

Builds OVP platform.

optional arguments:
-h, --help            show this help message and exit
-tasks <TASKSFILE>   File that contains the application description.
-procs <PROCSFILE>   File that contains the processor descriptions.
-map <MAPFILE>       Mapping file.
-number_of_flits <NUMBER_OF_FLITS>
                        Variable needed to build header.
usage: build_header.py [-h] -tasks <INPUTFILE> [-procs <INPUTFILE>]
[-number_of_flits <NUMBER_OF_FLITS>] -map <MAPFILE>

Builds header file that contains the applications' macros.

optional arguments:
-h, --help            show this help message and exit
-tasks <INPUTFILE>   File that contains the application description.
-procs <INPUTFILE>   File that contains the processor descriptions.
-number_of_flits <NUMBER_OF_FLITS>
                        Variable needed to build header.
-map <MAPFILE>       Mapping file.
oliver@rothaus:~/mestrado$

```

Figure 4.16: XYCPRO help command.

5. EXPERIMENTS

With the aid provided by the tools implemented in this work it is possible to exploit a big range of scenarios. The exploration can be taken aiming better mapping alternatives or even evaluating and comparing different mappings or systems' configurations with the same task set but with parameters or target architecture variations.

Based on the flexibility of the tools and on the design space flexibility, three different types of experiments were driven. Each set of experiments is shown in Sections 5.1, 5.2 and 5.3.

The former set of experiments tries to explore target architecture characteristics such as throughput, area and complexity. To do that, simulation and synthesis tools were used to obtain some metrics. The simulations performed were done by instruction set simulators and they compared buses, mesh-based and cluster-based NoCs in ideal conditions. The next type of experiments also considers simulations in instruction level, but instead, simulations try to explore the performance of different architectures when the communication media suffers from the injection of many concurrent data injections. The last type of experiments shows results obtained using the implemented mapping tools.

5.1 Architecture Analysis

To analyse the implemented architectures and their characteristics some configurations were used and instruction-level simulations were used to extract the cost of sending and receiving messages. Each message is sent through the media exclusively, i.e. the communication infrastructure does not suffer from effects of contention once there is only one message being transmitted at a time.

The task set used is very simple and has only two types of tasks. The core identified by ID 0 runs exclusively the task *Send*. This task sends a message and waits for an *ack* message. When it receives the *ack* message it calculates the time needed to send the original message and receive the *ack*. This process is repeated to each processing element attached to the architecture and the time calculated is annotated. See Figure 5.1 for more details about the *Send* task. The remaining processing elements of the architecture are mapped with the *Receive* task. This task merely receives a message and sends back an *ack* message.

The first architecture simulated was a bus interconnecting 64 processing elements. All the processing elements are Plasma Processors and they run at 100MHz while the bus's frequency was set to 6MHz. This decision was taken because of the relative good throughput of the architecture that difficults the elaboration of scenarios. Actually this decision will become sensitive only in the next type of experiments applied. In addition, the network interfaces are configured with FIFOs with 128 positions and each flit is 16 bits long. FIFOs with such size were chosen because messages are

```

1: function Send()
2: for  $i \in [1 - 63]$  do
3:    $cycles \leftarrow MemoryRead(COUNTER\_REG)$ 
4:    $SendMessage(i, 4096)$ 
5:    $ReceiveAck()$ 
6:    $time \leftarrow MemoryRead(COUNTER\_REG) - cycles$ 
7:    $StoreTime(i, time)$ 
8: end for

```

Figure 5.1: *Send* task description.

big (4096 bytes) and, therefore, there is less overhead by packet header. Consequently, less packets are needed to send each message.

The second architecture simulated is a mesh network-on-chip whose height is 8 and width is also 8. Frequencies configuration are the same used previously. Every tile has a processing element attached to its local port, so there are 64 processing elements too. Routers have incoming buffers whose depth is of 16 positions.

The last architecture is a clustered network-on-chip whose width, height and cluster width are two, four and eight respectively. Routers have incoming buffers whose depth is of 16 positions, while cluster interfaces have incoming and outgoing buffers of 128 positions.

Table 5.1: Bus (64) time costs.

Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
-	3687.27 μs	3687.76 μs	3687.76 μs	3687.92 μs	3687.92 μs	3688.09 μs	3688.1 μs
Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
3688.26 μs	3688.26 μs	3688.42 μs	3688.25 μs	3688.26 μs	3688.42 μs	3683.3 μs	3683.3 μs
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23
3683.47 μs	3683.47 μs	3683.63 μs	3683.64 μs	3683.8 μs	3683.8 μs	3683.96 μs	3683.97 μs
Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31
3684.13 μs	3684.13 μs	3684.3 μs	3684.29 μs	3684.46 μs	3684.46 μs	3684.63 μs	3684.63 μs
Core 32	Core 33	Core 34	Core 35	Core 36	Core 37	Core 38	Core 39
3684.79 μs	3684.79 μs	3684.95 μs	3684.96 μs	3685.12 μs	3685.12 μs	3685.29 μs	3685.28 μs
Core 40	Core 41	Core 42	Core 43	Core 44	Core 45	Core 46	Core 47
3685.45 μs	3685.45 μs	3685.62 μs	3685.62 μs	3685.78 μs	3685.79 μs	3685.94 μs	3685.95 μs
Core 48	Core 49	Core 50	Core 51	Core 52	Core 53	Core 54	Core 55
3686.11 μs	3686.11 μs	3686.12 μs	3686.27 μs	3686.28 μs	3686.44 μs	3686.44 μs	3686.61 μs
Core 56	Core 57	Core 58	Core 59	Core 60	Core 61	Core 62	Core 63
3686.61 μs	3686.77 μs	3686.78 μs	3686.94 μs	3686.93 μs	3687.1 μs	3687.1 μs	3687.27 μs

The time annotations of each scenarios are presented by Tables 5.1, 5.2 and 5.3. In Table 5.1, the times vary from 3683 μs to 3688 μs . See that the variation of time to send packets using

bus is of 5 μs . We will see that this variation is bigger than the NoC that has a variation time of only 2 μs (from 3662 μs to 3664 μs). We will see also that the cost to send data using NoC is smaller than the cost to send using bus in every situation. These phenomena were caused by implementation differences. The first difference are the buffers. In the NoC, once the flit is inserted into the Network Interface it can be forwarded through the NoC local port due to the existence of the FIFOs on the local port. This is not the case on the bus. Another difference is the arbitration logic delay. In the case of these two scenarios, it is less complex to perform the arbitration on multiple smaller router ports than arbitrate one bus with too many request ports.

Table 5.2: NoC (8x8) time costs.

Core 0 -	Core 1 3662.33 μs	Core 2 3662.49 μs	Core 3 3662.66 μs	Core 4 3662.82 μs	Core 5 3663.15 μs	Core 6 3663.15 μs	Core 7 3663.32 μs
Core 8 3662.33 μs	Core 9 3662.49 μs	Core 10 3662.5 μs	Core 11 3662.82 μs	Core 12 3662.99 μs	Core 13 3663.15 μs	Core 14 3663.32 μs	Core 15 3663.49 μs
Core 16 3662.49 μs	Core 17 3662.66 μs	Core 18 3662.66 μs	Core 19 3662.82 μs	Core 20 3663.16 μs	Core 21 3663.32 μs	Core 22 3663.48 μs	Core 23 3663.65 μs
Core 24 3662.66 μs	Core 25 3662.83 μs	Core 26 3662.99 μs	Core 27 3663.15 μs	Core 28 3663.32 μs	Core 29 3663.49 μs	Core 30 3663.49 μs	Core 31 3663.82 μs
Core 32 3662.83 μs	Core 33 3662.99 μs	Core 34 3663.16 μs	Core 35 3663.15 μs	Core 36 3663.48 μs	Core 37 3663.48 μs	Core 38 3663.82 μs	Core 39 3663.98 μs
Core 40 3662.99 μs	Core 41 3663.15 μs	Core 42 3663.16 μs	Core 43 3663.49 μs	Core 44 3663.65 μs	Core 45 3663.81 μs	Core 46 3663.98 μs	Core 47 3664.14 μs
Core 48 3663.15 μs	Core 49 3663.32 μs	Core 50 3663.32 μs	Core 51 3663.65 μs	Core 52 3663.82 μs	Core 53 3663.82 μs	Core 54 3664.14 μs	Core 55 3664.31 μs
Core 56 3663.32 μs	Core 57 3663.49 μs	Core 58 3663.65 μs	Core 59 3663.82 μs	Core 60 3663.98 μs	Core 61 3664.14 μs	Core 62 3664.14 μs	Core 63 3664.48 μs

Another interesting phenomenon that was expected and can be seen in Table 5.2 is the proportional increase of the calculated time as message is sent to more distant elements. For instance, to send message from Core 0 to Core 1 it takes 3662.33 μs while the same message takes 3664.48 μs to be sent to Core 63.

The last observations to make are related to clustered NoC. See that the cost to communicate between nodes in the same cluster is virtually the same as the cost to send using bus. However, there is less variation. For instance, the time cost to communicate in bus varies in 5 μs , while the variation using a cluster with 8 elements is less than 1 μs . This difference is caused by the arbitration logic that is less costly to arbitrate 8 elements instead of 16. Another thing to notice is that, as expected, messages changed between nodes in the same cluster cost less time to finish than messages whose nodes are attached to different clusters. For instance, send message to nodes in cluster 0 take times from 3683.07 μs to 3683.73 μs , while to send messages to cluster 7 takes from 3708.65 μs to 3709.96 μs .

Table 5.3: Clustered NoC (2x8x4) time costs.

Core 0 -	Core 1 3683.24 μs	Core 2 3683.07 μs	Core 3 3683.24 μs	Core 4 3683.56 μs	Core 5 3683.73 μs	Core 6 3683.24 μs	Core 7 3683.57 μs
Core 8 3706.84 μs	Core 9 3706.34 μs	Core 10 3706.51 μs	Core 11 3706.34 μs	Core 12 3705.67 μs	Core 13 3706.67 μs	Core 14 3706.01 μs	Core 15 3707.0 μs
Core 16 3705.68 μs	Core 17 3705.84 μs	Core 18 3706.0 μs	Core 19 3706.17 μs	Core 20 3706.34 μs	Core 21 3705.68 μs	Core 22 3705.84 μs	Core 23 3706.0 μs
Core 24 3707.65 μs	Core 25 3707.82 μs	Core 26 3707.33 μs	Core 27 3707.49 μs	Core 28 3707.65 μs	Core 29 3707.82 μs	Core 30 3707.16 μs	Core 31 3707.32 μs
Core 32 3706.83 μs	Core 33 3707.0 μs	Core 34 3707.16 μs	Core 35 3706.51 μs	Core 36 3707.49 μs	Core 37 3706.84 μs	Core 38 3707.82 μs	Core 39 3707.16 μs
Core 40 3708.15 μs	Core 41 3708.32 μs	Core 42 3708.31 μs	Core 43 3709.3 μs	Core 44 3708.64 μs	Core 45 3708.16 μs	Core 46 3708.97 μs	Core 47 3708.48 μs
Core 48 3707.16 μs	Core 49 3706.67 μs	Core 50 3708.15 μs	Core 51 3707.66 μs	Core 52 3707.82 μs	Core 53 3707.99 μs	Core 54 3707.49 μs	Core 55 3707.66 μs
Core 56 3708.49 μs	Core 57 3708.65 μs	Core 58 3709.47 μs	Core 59 3708.98 μs	Core 60 3709.14 μs	Core 61 3709.96 μs	Core 62 3709.47 μs	Core 63 3708.81 μs

5.1.1 Other Constraints

In addition to the system performance and latency, there are other constraints that may lead designer decisions. Some projects have area and power requisites, for example. With this context in mind, a simple study was driven to obtain some information about these design requisites.

Table 5.4 shows a comparison of NoC and clustered NoC with 64 processing elements that takes into consideration the requisite of area. The technology used to synthesize was 65nm from STMicroelectronics. It is clear to affirm that the cluster approach has distinguished advantages in this scenario. However, it is still not possible to affirm that cluster is always less consuming in terms of area than mesh NoC. To do such affirmation, more elaborated study should be performed.

Table 5.4: Area comparison between NoC and clustered NoC.

Architecture	Area (μm^2)	Growth
Clustered NoC (2 × 4 × 8)	388567	
NoC (8 × 8)	1313130	$\cong 338\%$

Table 5.5 makes a comparison of NoC and clustered NoC with respect to frequency. The objective was to synthesize the architectures with the goal to better explore frequency. The table shows that there is a decline on the cluster approach due to the growth of its logic complexity.

Table 5.5: Frequency comparison between NoC and clustered NoC on Virtex V. Adapted from [Mag13].

Architecture	Frequency	Decline
NoC(8×8)	167.805 MHz	
Clustered NoC($3 \times 3 \times 8$)	155.109 MHz	$\cong 8\%$

The two experiments shows that there is a gain on area but there is a loss on frequency. The next set of experiments will show that in some situations the throughput will decrease with cluster but the loss is so small and the gain in area is so big that it's worth choosing clusters instead of mesh networks.

5.2 System Simulation

The second type of experiments were done to elaborate an comparison of the architectures when these architectures are suffering from multiple injections of packets simultaneously. In such situations, the existence of contention is possible, and in fact, applications were intentionally selected to cause it.

The two applications explore pairs of communicating processing elements in very different aspects. The only difference on the hardware used was on the clustered NoC that used a configuration of $4 \times 4 \times 4$, i.e., NoC width. NoC height and cluster size were all set to 4.

The former application has tasks that communicate with the neighbour tiles. It means that tasks from node 0 communicate with tasks from node 1, tasks from node 2 communicate with tasks from node 3, and so on. Each node is mapped with two tasks, one task identified as *Send* and another as *Receive*. There are two types of messages transferred: data and acknowledge messages. Data messages have 4kb while *ack* messages have only a single packet of size. Packets are composed by 128 flits, consequently network interface buffers have 128 positions too.

The other application is almost the same as the previous one but with one exception. The communications occur with more distant nodes. For instance, in this scenario, tasks placed on node 0 would communicate with tasks placed on node 4 and tasks from node 1 would communicate with tasks mapped to node 5, and so on. Of course, this configuration does not differ when the bus communication media is used, but an appreciable difference can be viewed when other media topologies are used.

Every scenario were simulated for 100.000.000 processor instructions. Plasma processors operating in 100 MHz were used. The routers were configured with 6 MHz of operating frequency. As mentioned earlier, this decision was made to highlight the contention of the architecture. Another way to do it could be increasing the number of processors. However, the alternative to decrease the media operating frequency was chosen to ease the design elaboration and avoid simulation times too long.

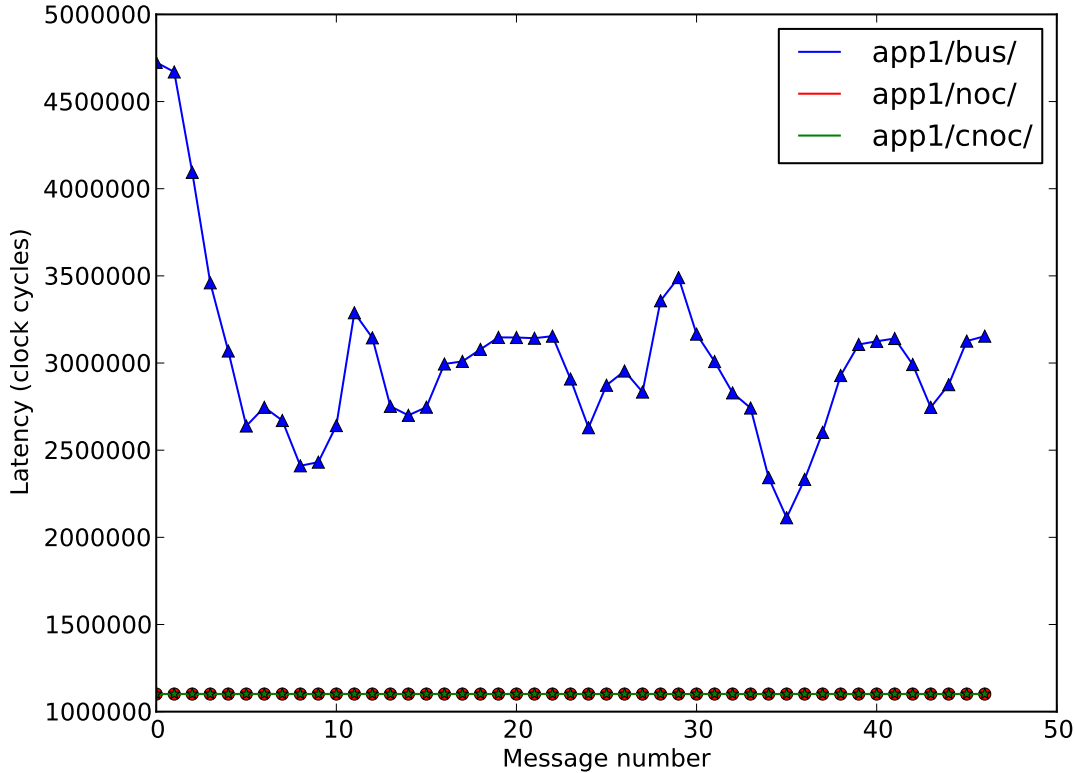


Figure 5.2: Performance analysis of application 1. See that line plots for NoC and CNoC are difficult to distinguish due to their proximity.

The time to send messages of the first application were annotated and are shown in Figure 5.2. See that only 47 messages were shown due to the fact that bus could not perform more communications in the amount of instructions simulated.

There are two things to highlight immediately. The Bus is definitely the media whose duration times are longer and vary mostly. In addition, the times annotated for NoC and clustered NoC are virtually the same. Actually mesh average duration time is shorter but the same reasons given on previous experiments (NoC buffers and distributed arbitration enables faster communications) applies in this case as well.

Figure 5.3 shows the annotated times for the second application. As explained earlier, the tasks of this application communicates with tasks mapped to more distant nodes. Such characteristics are better handled by architectures that offers more parallelism. In the case of the architectures used, it is clear to assert that the network-on-chip offers more parallelism, thus, have the shorter duration times. On the other hand, cluster-based NoC takes longer times but is still significantly more efficient than bus.

See that bus can still perform only 47 communications per node during the simulated time. The number does not vary owing to the fact that message addressing (that changes from application one to application two) makes no effect on the bus. You can see that in Figure 5.4, that plots annotated times for both applications.

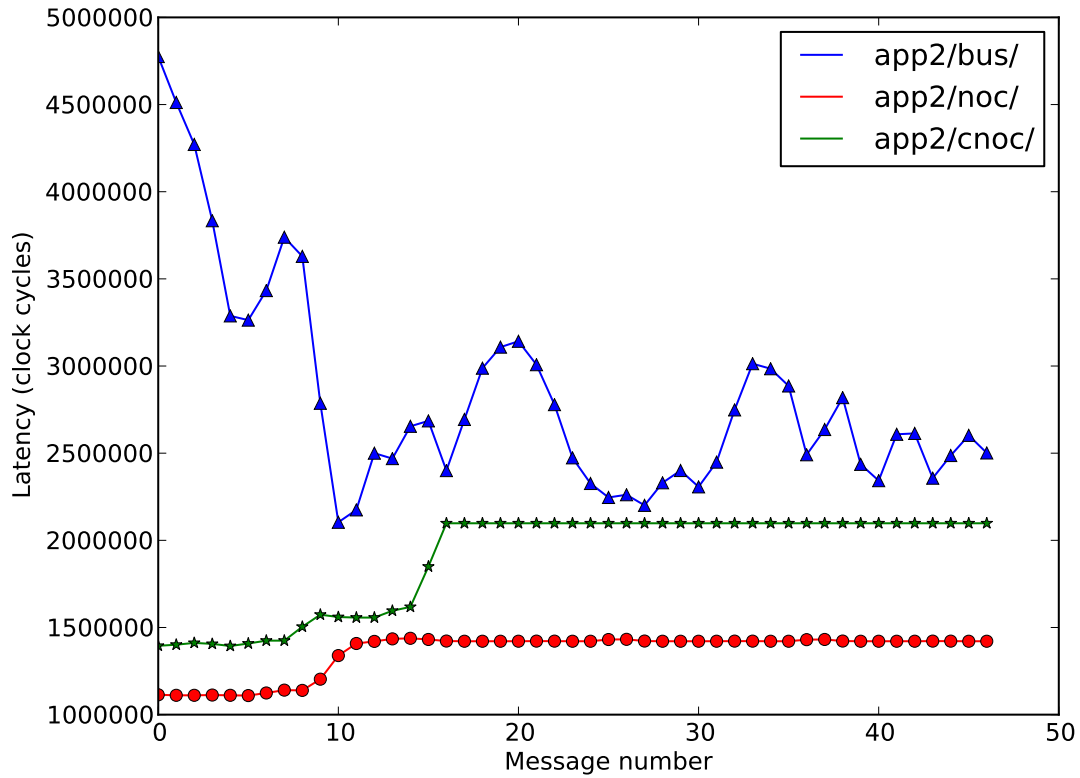


Figure 5.3: Performance analysis of application 2.

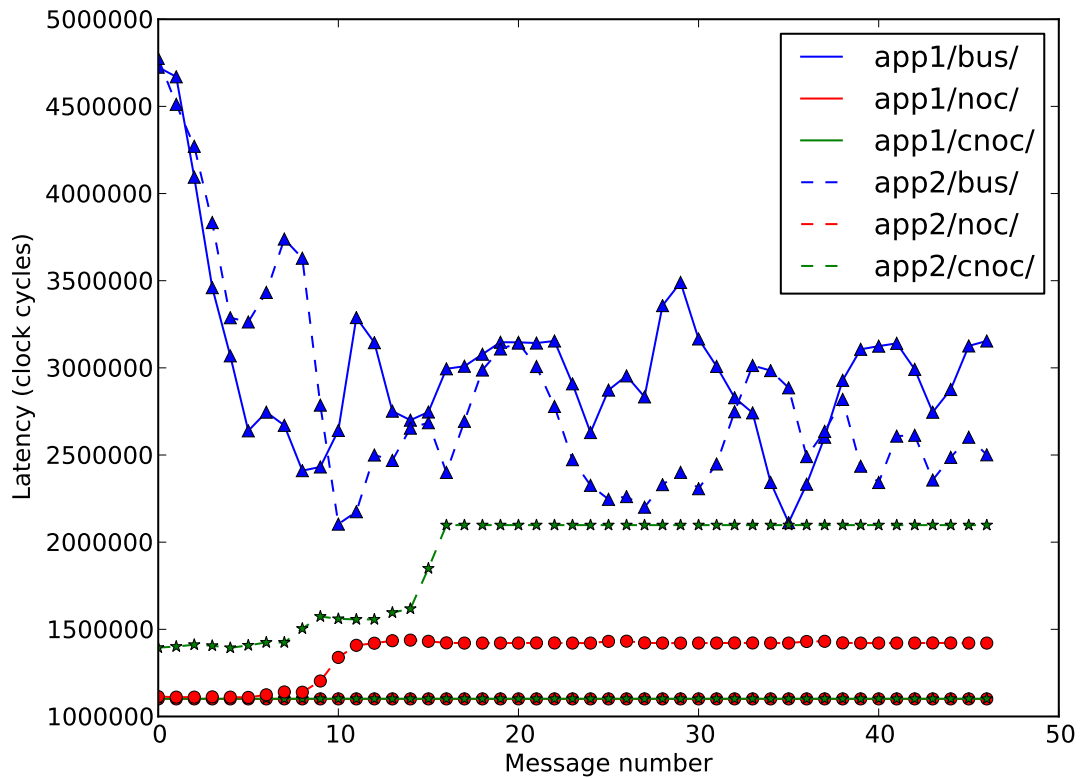


Figure 5.4: Performance analysis of applications 1 and 2.

There is another phenomenon that may call reader's attention. In Figure 5.3 it is possible to see a variation of the communication durations in the 20 initial messages. Actually this variation exists too in the application one but can be better seen in this second scenario. The cause of this variation is the deparallelization of the communications. As more contention is created by the injection of packets into the communication media and as the operating systems schedules tasks in a more sparse manner, less dispute for accessing the media occurs. Consequently, less contention is originated. It makes duration time on bus decrease and the parallelism of NoCs be less enjoyed.

5.3 Mapping Algorithms

Another type of experiments were driven to evaluate the mapping tools. We applied the implemented abstract models with different configurations of architectures. The architectures used were: (a) Network-on-Chip whose both height and width are 4, and (b) a cluster-based Network-on-chip whose height and width are 2, and each router links clusters with length of 4 processors. The clusters are implemented with buses and the communication model is message passing.

The applications implemented are: (a) Video Object Plane Decoder [MDM04], that is a multimedia processing application composed by 17 communicating tasks; (b) MPEG4, a Video Encoder described in [MMV07] that comprehends 18 communicating tasks; and (c) MJPEG, an image compressor with 4 pipelined tasks.

It is worth remembering that the meaning of the returned *Mapping Cost* is how many bits (*in the worst case*) are necessary to be forwarded before a message can be delivered. Therefore, the absolute value returned should not be taken by itself as a meaning value because it is based on a worst case scenario. However, it is still useful to compare different topologies.

In Figures 5.5 and 5.6 the execution trace of the three configurations are shown. The convergence curve of the third implementation of the Simulated Annealing (SAN2) stands out when compared to the others. It happens due to the fact that the acceptance test of the simulated annealing occurs less frequently. Executing less acceptance tests in the initial iterations is an important characteristic because at that point the temperature variable is still configured with a high value. Consequently, it is less likely to accept alternative mapping costs that increase the system's latency.

It is possible to see that the solutions' results found for architectures 4×4 and $2 \times 2 \times 4$ are very similar and the NoC has a performance slightly better in most of the cases. This happens because architectures implemented with regular NoC can better explore the parallelism but does not gain benefits from local message communications since there is only one processor linked to each router. See that this phenomenon is very similar to the results obtained in the two previous types of experiments where the cluster architecture performances were very similar but always slightly smaller than the NoC.

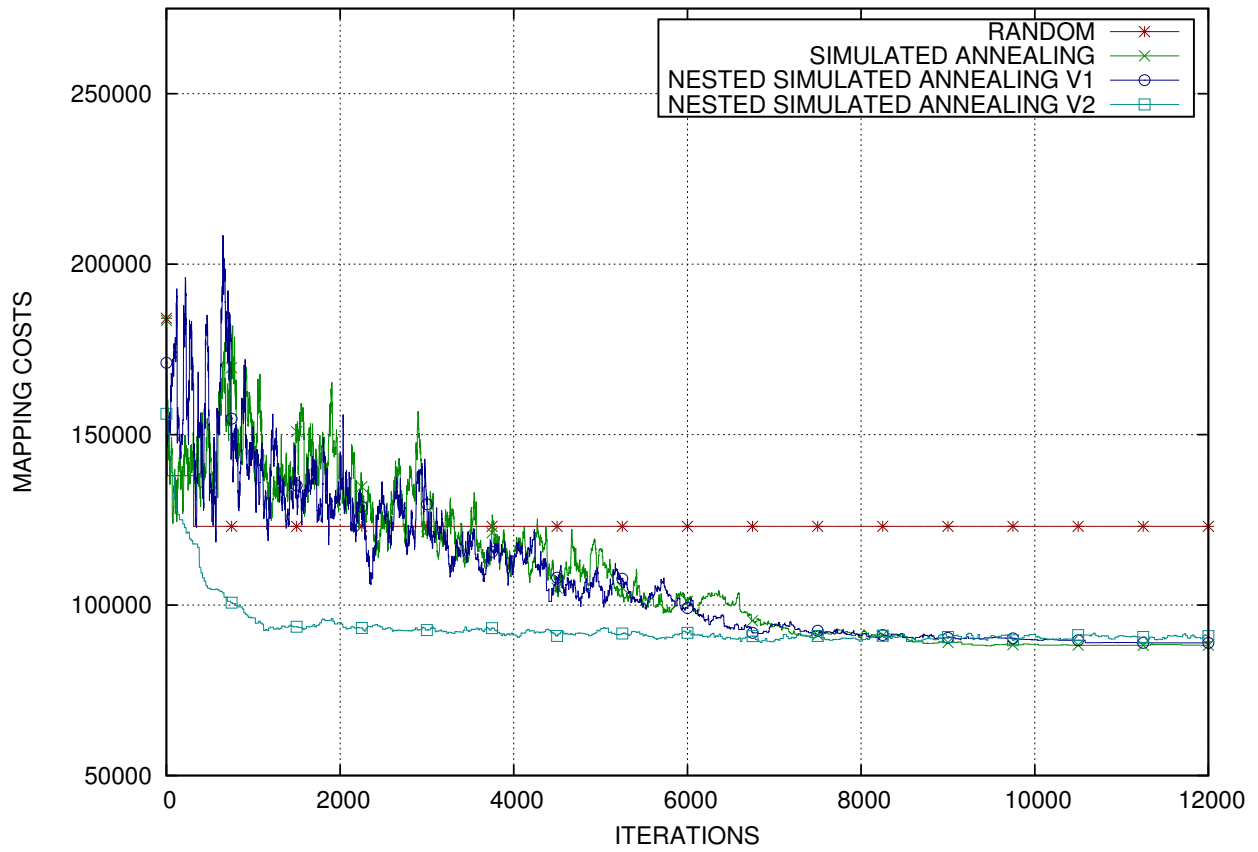


Figure 5.5: Mapping costs for the 4×4 architecture.

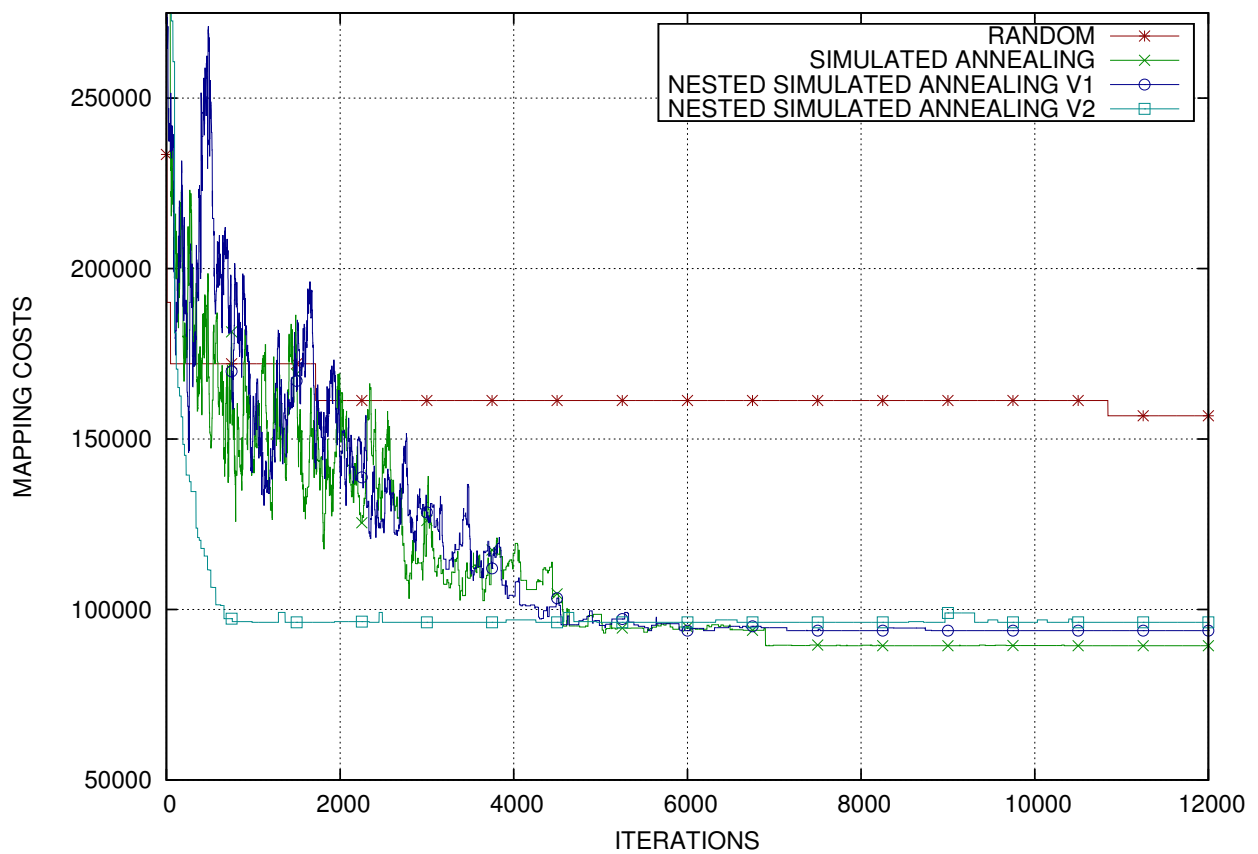


Figure 5.6: Mapping costs for the $2 \times 2 \times 4$ architecture.

The whole set of experiments shows that clustered architectures is not an unbeatable approach. The performance of clustered architecture is very sensitive to the application communication, hence tasks mapping is also very sensitive to the application performance. Basically, we have seen that mappings, whose communicating tasks are not on the same cluster, use to be very burdened. Additionally, due to intrinsic characteristics of local arbitration and hardware complexity, it is difficult to find mappings where the overall system performance of a cluster based platform is bigger than the performance of mesh based platforms. In particular, the results shown that in the best case, clustered architectures have virtually the same (or an insignificant lower) throughput.

6. FINAL CONSIDERATIONS

There are many considerations to take based on the experiments elaborated. The former concerns about the fact that many applications are not well supported by clustered architectures. We have seen that communication between non-neighbour nodes can lead to very inefficient architectures. On the other hand, it was also possible to see that clusters can decrease significantly other hardware constraints and still have the same throughput as a mesh-based network-on-chip. In particular, the results shown that spite of the fact that there was a very small loss in terms of performance, the gains on area were extremely relevant.

Another important point that is important to highlight is the variation between the mapping algorithms and the simulation results. Actually, more effort have to be done to make high-level models represent properly low-level models. However, we cannot overlook the effort already done considering that the set of tools implemented in the work are of great value for their purposes. See that the work flow implemented demonstrates a high order of complexity and can be easily used to explore architectures, nevertheless adjustments and improvements are always feasible.

Future Work

There are many aspects that can be explored based on the work proposed. For instance, it would be of great value if the set of tools of the work flow could be abstracted by a design framework. The framework would have as input a task set and some target processors, and the output would be a ready-to-prototype architecture. Alternatively, some of the implemented tools could also be used to extend existing tools from third-party systems, like Hellfire Framework.

In relation to mapping algorithm, an important feature that could be added is the possibility to use multiple scheduling policies like Earliest Deadline First and any other hybrid scheduler that is also implemented by Hellfire Operating System.

Other important contribution of the work that can still evolve is the communication graph MCWM. It is a superset of the well-known CWM that, with support of simulators, enables a representation of communication behaviour with more details than its predecessor. Such evolution can happen by incorporating other details about communication, for instance, discriminating dependency or parallelism between communications.

Finally, one very interesting feature that could be added to the work is the possibility to represent 3D networks. To do that abstract and high-level models should be implemented but all the tools were designed aiming future modifications, so, it would not demand a big rework to achieve such extension.

BIBLIOGRAPHY

- [AFM⁺10] Aguiar, A.; Filho, S.; Magalhaes, F.; Casagrande, T.; Hessel, F. "Hellfire: A design framework for critical embedded systems' applications". In: Quality Electronic Design (ISQED), 2010 11th International Symposium on, 2010, pp. 730–737.
- [ANPV10] Avakian, A.; Nafziger, J.; Panda, A.; Vemuri, R. "A reconfigurable architecture for multicore systems". In: Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, 2010, pp. 1 –8.
- [BFFM12] Benini, L.; Flmand, E.; Fuin, D.; Melpignano, D. "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator". In: Design, Automation Test in Europe Conference Exhibition (DATE), 2012, 2012, pp. 983 –987.
- [BM06] Bjerregaard, T.; Mahadevan, S. "A survey of research and practices of network-on-chip", *ACM Comput. Surv.*, vol. 38–1, Jun 2006.
- [CS00] Christie, P.; Stroobandt, D. "The interpretation and application of rent's rule", *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 8–6, Dec 2000, pp. 639–648.
- [DT01] Dally, W.; Towles, B. "Route packets, not wires: on-chip interconnection networks". In: Design Automation Conference, 2001. Proceedings, 2001, pp. 684 – 689.
- [FYX⁺10] Fangfa, F.; Yuxin, B.; Xinaan, H.; Jinxiang, W.; Minyan, Y.; Jia, Z. "An objective-flexible clustering algorithm for task mapping and scheduling on cluster-based noc". In: Laser Physics and Laser Technologies (RCSLPLT) and 2010 Academic Symposium on Optoelectronics Technology (ASOT), 2010 10th Russian-Chinese Symposium on, 2010, pp. 369 –373.
- [GG00] Guerrier, P.; Greiner, A. "A generic architecture for on-chip packet-switched interconnections". In: Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings, 2000, pp. 250 –256.
- [HM03] Hu, J.; Marculescu, R. "Energy-aware mapping for tile-based noc architectures under performance constraints". In: Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific, 2003, pp. 233 – 239.
- [HYH⁺11] Huang, C.-Y.; Yin, Y.-F.; Hsu, C.-J.; Huang, T.; Chang, T.-M. "Soc hw/sw verification and validation". In: Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific, 2011, pp. 297 –300.
- [ITR09] ITRS. "The international technology roadmap for semiconductors (itrs)". Capturado em: <http://itrs.net>, 2009.

- [KGV83] Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. "Optimization by Simulated Annealing", *Science, Number 4598, 13 May 1983*, vol. 220, 4598, 1983, pp. 671–680.
- [KJS⁺02] Kumar, S.; Jantsch, A.; Soininen, J.-P.; Forsell, M.; Millberg, M.; Oberg, J.; Tiensyrja, K.; Hemani, A. "A network on chip architecture and design methodology". In: *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on, 2002*, pp. 105–112.
- [LK03] Lei, T.; Kumar, S. "A two-step genetic algorithm for mapping task graphs to a network on chip architecture". In: *Digital System Design, 2003. Proceedings. Euromicro Symposium on, 2003*, pp. 180–187.
- [LRD01] Lahiri, K.; Raghunathan, A.; Dey, S. "Evaluation of the traffic-performance characteristics of system-on-chip communication architectures". In: *VLSI Design, 2001. Fourteenth International Conference on, 2001*, pp. 29–35.
- [LSS⁺08] Lin, S.; Su, L.; Su, H.; Jin, D.; Zeng, L. "Hierarchical cluster-based irregular topology customization for networks-on-chip". In: *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on, 2008*, pp. 373–377.
- [Mag13] de Magalhães, F. G. "Hc-mpsoc : plataforma do tipo cluster para sistemas embarcados", 2013.
- [Mar05] Marcon, C. A. M. "Modelo para mapeamento de aplicações em infra-estruturas de comunicação intrachip", Ph.D. Thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brazil, 2005, 192p.
- [MBF⁺12] Melpignano, D.; Benini, L.; Flamand, E.; Jegou, B.; Lepley, T.; Haugou, G.; Clermidy, F.; Dutoit, D. "Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications". In: *Proceedings of the 49th Annual Design Automation Conference, 2012*, pp. 1137–1142.
- [MCM⁺04] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "Hermes: an infrastructure for low area overhead packet-switching networks on chip", *Integration, the {VLSI} Journal*, vol. 38–1, 2004, pp. 69–93.
- [MDM04] Murali, S.; De Micheli, G. "Bandwidth-constrained mapping of cores onto noc architectures". In: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings, 2004*, pp. 896–901 Vol.2.
- [MMCM08] Marcon, C.; Moreno, E.; Calazans, N.; Moraes, F. "Comparison of network-on-chip mapping algorithms targeting low energy consumption", *Computers Digital Techniques, IET*, vol. 2–6, november 2008, pp. 471–482.
- [MMV07] Milojevic, D.; Montperrus, L.; Verkest, D. "Power dissipation of the network-on-chip in a system-on-chip for mpeg-4 video encoding". In: *Solid-State Circuits Conference, 2007. ASSCC '07. IEEE Asian, 2007*, pp. 392–395.

- [MSA12] Modarressi, M.; Sarbazi-Azad, H. "Reconfigurable cluster-based networks-on-chip for application-specific mpsocs". In: Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on, 2012, pp. 153 –156.
- [OVP12] OVP. "Open Virtual Platform Website", 2012.
- [TLP+10] Tsai, K.-L.; Lai, F.; Pan, C.-Y.; Xiao, D.-S.; Tan, H.-J.; Lee, H.-C. "Design of low latency on-chip communication based on hybrid noc architecture". In: NEWCAS Conference (NEWCAS), 2010 8th IEEE International, 2010, pp. 257 –260.
- [TMT12] TANG, Q.; MEHREZ, H.; TUNA, M. "Design for prototyping of a parameterizable cluster-based multi-core system-on-chip on a multi-fpga board". In: Proceedings of the 2012 23rd IEEE International Symposium on Rapid System Prototyping, 2012, pp. 71 – 77.