Dynamic Process Relocation in Multiprocessor Systems

Mixed-Criticality Aware Implementation

TAGE MOHAMMADAT
Dynamic Process Relocation in Multiprocessor Systems

Mixed-Criticality Aware Implementation

Tage Mohammadat

Master of Science Thesis

Department of Electronics
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

23 March 2017

Supervisor: Johnny Öberg
Examiner: Ingo Sander
Abstract

Enabling deterministic dynamic process relocation in electronic systems creates opportunities for dynamic computational load balancing, fault tolerance, power consumption management, which can be positively correlated with improvements in longevity, safety and energy-efficiency. Additionally, dynamic process relocation can be leveraged to enhance the adaptability of mixed-critical systems operating in open and changeable environments, which is a current market driver for safety-concerned industries like avionics, automotive and railways. Future electronic systems in such industries are envisioned to comprise networked heterogeneous multiprocessor chips that execute applications with different criticality levels. With such setting, it is not clear from the literature how dynamic process relocation (DPR) can be implemented in a way that does not compromise predictability concerns of safety-critical functionalities in such mixed-critical system. In this work, a design and implementation framework is demonstrated that takes a list of inputs that satisfy DPR design constraints and exploits DPR design primitives, protocol and design flow to produce the required implementation. The design inputs include application communication graph, application to platform mapping, process criticality levels and DPR scenarios. The implementation is demonstrated on a system of two printed circuit boards, each containing monolithic heterogeneous dual-core ARM cortex A9 processor networked with Microblaze soft-processors via a mesh-topology time-triggered Nostrum network-on-chip (NoC). The work shows promising predictable adaptability suitable for mixed-critical systems and could lead to improved service efficiency and availability.
Sammanfattning

Deterministisk dynamisk processflyttning (eng. Dynamic Process Relocation - DPR) i elektroniska system skapar möjligheter för dynamisk beräkning av belastningsbalans, feltolerans, och strömförbrukningshantering, vilket kan ha en positiv korrelation med förbättringar i livslängd, säkerhet och energieffektivitet. Dessutom kan dynamisk processflyttning utnyttjas för att förbättra anpassningsförmågan av mixed-critical system i öppna och föränderliga miljöer, vilket är en nuvarande marknadskrav inom säkerhetsrelaterade teknikområden såsom flygelektronik, fordonsin industri och järnvägar. Framtida elektroniska system i sådana industrier förväntas bestå av nätverk av heterogena multiprocessorchips som kör program med olika säkerhetskritiska nivåer. Det framgår det inte av litteraturen hur dynamisk processflyttning (DPR) i ett sådant scenario kan genomföras på ett säkert sätt och inte samtidigt inte äventyrar försägbarheten av säkerhetskritiska funktioneliteter. Denna rapport demonstrerar upp ett fungerande designflöde, vilket utgår från en lista av input som uppfyller DPR-design begränsningar och utnyttjar DPR design primitiv, protokoll och designflöde för implementeringen. Dessa design inputs inkluderar applikations och kommunikationsgrafer, applikation till plattforms mappings, processkritiska nivåer och DPR-scenarier. Implementeringen demonstreras på ett system bestående av två kretskort /PCB:er, som vardera innehåller en FPGA med en heterogen design bestående av en ARM-cortex dual-core A9-processor, vilken är ansluten till 3 st mjuka processorer av Microblaze typ, via ett tids-trigged Network-on-chip (NoC) av Nostrum Mesh topologi typ. Arbetet visar lovande förutsägbarhet för DPR av mixed-kritiska system, vilket kan leda till förbättrad effektivitet och ökad tillgänglighet av service.
Acknowledgements

I would like to thank my colleagues Henrik Woidt, Kalle Ngo, Joanna Alexander, Dimitrios Stathis and Matteo Birmini for being stimulating, supportive and entertaining during the course of this work. Special thanks to Youssef Zaki for being collaborative, complementing and constructive in the journey of the degree project and the master programme generally.

During this journey I have benefited from side discussions and insights from Nasim Farahini, Nikola Ivanisevic and Sha Tao to whom I feel thankful.

I would also like to acknowledge the significant contributions of certain professors at KTH Royal Institute of Technology whose efforts have undoubt-edly positioned me in a good proximity to the state-of-the-art by either providing mentor spaces, great insights into embedded electronic system design, stimulating ideas and opportunities or scholarly tips on research. This includes: Johnny Öberg, Ingo Sander, Ahmed Hemani, Mark Smith, Ana Rusu, Elena Dubrova and Gerald Q. (Chip) Maguire Jr.

The whole idea of doing this master programme and the experience of studying at Kungliga Tekniska Högskolan would have not been possible without the scholarship grant of Svenska Institutet, to which I’m grateful.
# Contents

1 Introduction ........................................ 1
  1.1 Problem Context .................................. 2
  1.2 Goals ............................................ 5
  1.3 Project Contextualisation ......................... 5
  1.4 Scope and Limitations ............................. 6
  1.5 Structure of this Document ....................... 8

2 Background ........................................... 9
  2.1 Mixed-Criticality Systems ......................... 9
  2.2 Multi-Processor System-on-Chips .................. 10
  2.3 Network-on-Chip Architecture ..................... 11
  2.4 System Design Automation ......................... 14

3 Design and Implementation .......................... 17
  3.1 Overview ......................................... 17
  3.2 Design Assumptions ................................ 21
  3.3 Design Framework Primitives ....................... 23
    3.3.1 Master NI .................................... 23
    3.3.2 Process Handler ............................... 27
    3.3.3 Network Channels Allocation .................. 28
    3.3.4 Dynamic Process Relocation Protocol .......... 31
  3.4 Implementation .................................... 32
    3.4.1 Potential Scenarios ........................... 32
    3.4.2 Implementation Flow ........................... 34

4 Results and Discussion .............................. 43
  4.1 Results ........................................... 43
  4.2 Discussion ........................................ 45
    4.2.1 Potential Energy Consumption Reduction ...... 46
    4.2.2 Potential Reliability Improvement ............. 46
    4.2.3 Design Optimisation ........................... 47
5 Concluding Remarks
  5.1 Contributions ........................................... 50
  5.2 Future directions ........................................ 50

Bibliography ................................................. 51

Appendices .................................................. 61

A Intellectual Property Blocks ........................................ 61
  A.1 Embedded Processor Nodes .................................. 61
  A.2 Network-on-Chip IP Blocks .................................. 61
  A.3 Design Automation .......................................... 64

B NoC Synthesis and Characterisation ................................. 65
# List of Figures

1.1 A mixed-criticality system unit breakdown ............... 3  
1.2 Problem context ........................................ 4  
2.1 An example of a multi-processor system on a chip ........ 11  
2.2 An example of a network on a chip ....................... 12  
3.1 Illustration of process handlers and network daemon ....... 19  
3.2 Design flow ............................................. 21  
3.3 Dynamic process relocation example ....................... 22  
3.4 Network daemon tasks .................................... 25  
3.5 Network daemon and proxy communication ................ 26  
3.6 DPR instruction set ..................................... 26  
3.7 DRP instruction format ................................... 26  
3.8 Process handlers’ tasks ................................... 27  
3.9 General design idea ..................................... 30  
3.10 Dynamic process relocation protocol ..................... 31  
3.11 Implementation scenarios ................................ 33  
3.12 Network daemon and process handlers communication .... 34  
3.13 Processes at chip 1 ....................................... 35  
3.14 Processes at chip 2 ....................................... 36  
3.15 Enumeration of processes and communication channels .... 37  
3.16 Process equivalence table example ....................... 39  
3.17 Process status table example ............................ 39  
3.18 Construction of decision trees for packet forwarding .... 40  
3.19 Comprehensive dynamic process relocation decision trees . 41
List of Tables

3.1 DPR table ................................................. 33
3.2 Channel allocation example ............................... 38
4.1 DPR Overhead ............................................ 43
A.1 Network interface (NI) configuration ................. 62
A.2 Switches network configuration ....................... 62
A.3 List of IPs used in the system implementation ........ 63
A.4 Versions of tools used in the design and development .... 64
B.1 NoC timing parameters ................................. 66
B.2 Resource utilisation ................................ 66
B.3 Performance metrics .................................. 67
# List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer-Aided Design</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DPR</td>
<td>Dynamic Process Relocation</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multi-Processor System-on-Chip</td>
</tr>
<tr>
<td>NI</td>
<td>Network Interface</td>
</tr>
<tr>
<td>NoC</td>
<td>Network-on-Chip</td>
</tr>
<tr>
<td>SDA</td>
<td>System Design Automation</td>
</tr>
<tr>
<td>SLS</td>
<td>System-Level Synthesis</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Reducing the number of electronic processing units to reduce cabling, weight and consequently reducing fuel consumption and improving reliability is a contemporary market driver for automotive, avionics and railway industries [1, 2, 3]. For example, in the automotive industry, a 20% reduction of the number of electronic control units (ECUs) is a desired target for future electronic sub-systems [1]. This could be achieved by exploiting fewer but computationally-intensive multiprocessor-based ECUs. Systems resulting from the integration of software processes of different levels of criticality onto the same hardware unit is known in the literature as mixed-criticality systems [4]. Such systems have to be designed such that the execution of critical services is always guaranteed. On the other hand, less critical process execution may be optimised during run time by providing the available resources depending on the operating conditions and imposed design constraints like power consumption and die temperature. Additionally, meeting industry requirements such as dynamic adaptability to on-demand run-time services without violating service guarantees for critical processes introduce further design challenges on the run-time flexibility of the system.

Multi-processor system-on-chip (MPSoC) based platforms, especially those comprising monolithic implementation of heterogeneous processors, peripherals and run-time recon-figurable field programmable gate array (FPGA), are promising platforms for mixed criticality system implementation. This is due to the potential advantage of the hybrid hard and soft Intellectual Property (IP) blocks within the MPSoC platform. For instance hardened processor and peripherals within MPSoCs are not only more energy-efficient, but also could run tasks as fast as five times faster than their counter-parts implemented on pure FPGA [5]. On the other hand, the exploitation of the tightly coupled FPGA part of the MPSoC can speed-up performance to an order of magnitude more than a pure general-purpose ASIC processor
Chapter 1. Introduction

In many-core multiprocessor systems, point-to-point and bus-based interconnection architectures become slow or energy inefficient. Instead, a network-on-chip (NoC) is deemed more scalable [9]. This is attributed to the associated electrical capacitive load effects and arbitration latency of a bus-based system or the large number of interconnections in point-to-point communication [10]. Additionally, a NoC based communication can provide a trade-off between security due to non-shared memory space, and energy-efficiency due to partial network resource sharing, compared to a bus-based shared-memory and a point-to-point communication mechanism [11].

Dynamically reconfigurable MPSoCs can enable low-power, computationally intensive and dynamic adaptability [12, 13, 14]. However for dynamic adaptability to function correctly in safety critical systems, all system components, including interprocessor communication, should be designed to cope with the possible dynamic alteration to the system.

In this work, a dynamic adaptability in terms of process relocation is considered. The focus will be on developing design and implementation framework around a typical NoC to cope with dynamic process relocation. A MPSoC comprises an ASIC processor such as dual core ARM-A9 [15, 16], and FPGA-based soft-cores such as Xilinx µBlaze [17] is considered the target implementation platform. The implementation will be made on two networked MPSoC boards to demonstrate potential design scalability which is a typical industrial requirement. The specific contribution is the provision of a design that supports the dynamic relocation of processes in a predictable manner. This work may be considered as a step towards adaptive design methodology that enables local-cloud like computation.

1.1 Problem Context

In a simplified and lumped view, the single unit of a mixed-criticality system can be a MPSoC which can be abstracted as in Fig. 1.1. The figure portraits the unit as being consisting of four main components: processing engines (µP) to compute algorithms, memory where algorithms and data reside, communication infrastructure to enable processing engines to exchange messages for the functionality of the system, and peripherals such as input and output devices, sensors and actuators to interface with the physical world. A mixed-criticality platform can consist of several MPSoCs communicating via an interconnection network. Fig. 1.2 shows a typical part of a mixed-criticality system where processes of different
criticality are present on the same networked MPSoC platform. In this view an interconnection network is one of the system components to enable communication and on which the correct and predictable message delivery of inter-processor communication considering dynamic process relocation scenario should be supported.

Figure 1.1: A mixed-criticality system unit breakdown and its major components

Despite the advances in MPSoC technologies and mixed-criticality design research, it is not entirely clear how run-time adaptability such as dynamic process relocation (DPR) can be achieved in a deterministic manner in the context of mixed-criticality systems. Moreover, the mechanisms required for processor interconnection network to be capable to support dynamic adaptability in terms of dynamic process relocation (DPR) for mixed-criticality systems have not been explicitly addressed.
Chapter 1. Introduction

Local Network of Region/Chip Y

Local Network of Region/Chip X

How would the NoC know new process destination?

How messages fidelity can be guaranteed in mixed criticality context?

To support:

- Fault tolerance
- Longevity improvement
- Energy reduction

Network to Network Bridge

Figure 1.2: Problem context: the figure poses a question of how dynamic process relocation should be supported to enable efficient execution of expensive and computationally-intensive services, distribute processing loads and manage hot-spawning in the context of networked real-time mixed-critical NoC-based MPSoCs. In the diagram, critical and non-critical processes are indicated by red and blue circles respectively, connected dashed circles indicate relocatable processes.
1.2 Goals

This work attempts to address the afore-mentioned problems by providing an implementation framework with the following specific contributions:

- Proposal of design primitives, methods, protocol and software services for process synchronisation and relocation on processing nodes.

- Demonstration of the potential of this framework by implementing DPR support on a system of NoC-based MPSoCs.

In the demonstration, a system consisting of two Zedboard boards [18] is considered for the implementation due to the presence of a MPSoC device and communication peripherals such as CAN and Ethernet for chip-to-chip communication. Within each MPSoC, a two by two mesh Nostrum-based NoC, connecting four computational subsystems, is considered. The processing or computation subsystems are based on Xilinx Microblaze microprocessors and the dual ARM-core processor. Each processing subsystem can have several software processes, or tasks or services, that communicate with other processes residing within the network or outside the network. The work goals can be summarised as follows:

- Designing and implementing a framework considering 2x2 Nostrum NoC-based MPSoC on a Xilinx Zynq device such that dynamic process relocation (DPR) is supported.

- Demonstration of dynamic process relocation over two interconnected printed circuit boards.

1.3 Project Contextualisation

Technologies for non-volatile memory like flash and magnetic disks have orders of magnitude more capacity and order of magnitude more access latency compared to volatile DRAM, SRAM and D-Flip-Flops memory-addressable counterparts. It is possible to exploit this technological feature in run-time reconfigurable FPGA based system-on-chip hardware, to design systems whose subset of modules are mutually exclusive in execution. This can be achieved by allowing hardware configuration and its software to be stored within, or external to the system, and to be activated later at run-time using a configuration controller when it is required. If the time to reconfigure the hardware is tolerable by the system operation requirement, such approach can virtually be equivalent to a larger chip with all modules
loaded. For specific applications and low production volume in comparison to pure custom ASIC implementation, this approach has the potential of reducing static energy dissipation, reducing development time and costs and provide fault-tolerance capabilities if dynamic reconfiguration is leveraged for fault-detection and diagnosis, to provide redundancy, hardware relocation and mapping. Such operation scenario can be also considered as a form of local-cloud computation.

Multi-processor system-on-chips combine the advantages of ASIC based processor and run-time flexibility of FPGAs as it contains one or more hard processor(s) and tightly-coupled programmable logic to act as glue logic or hardware acceleration. This is advantageous in mixed-criticality system due to the potentiality of power-efficiency, design scalability and reduced interference between critical and non-critical processes – when correct mapping and scheduling is made. Nevertheless, integrating processes from multiple single-core chips onto a single multi-processor system on chip increases thermal intensity in a way that could affect the safe operation and energy efficiency, therefore deterministic dynamic management techniques might be employed to achieve better safety, energy-efficiency and longevity. Dynamic process relocation is a potential dynamic management technique for such systems.

In spite of the potential benefits, the state-of-the-art does not seem to be particularly suggesting design methods for implementing dynamic process relocation in mixed critical applications on multi-processor system-on-chips. This highlights a challenge this work aspires to overcome. It is envisaged that a design framework as such would enable further run-time flexibility to safety-critical systems. It can also help in achieving more power and fault management, and eventually safer operation.

1.4 Scope and Limitations

Several limitations, assumptions and constraints have been considered for a practical implementation of this work within the allocated time frame which includes:

- The base NoC is assumed to employ time-triggered NoC based on Nostrum Heartbeat NoC [19, 20, 21].

- All possibilities for dynamic process relocation is defined at design time to allow analysis on predictability, safety and real-time performance.

- The conditions upon which relocation occurs to allow optimal system performance will not be considered in this work.
1.4. Scope and Limitations

- The dynamic process relocation does not imply any hardware reconfiguration to the processing node.

- Only state-less periodic software processes are considered in the demonstration. This is not a realistic requirement but convenient and practical to simplify the demonstration of process relocation. Considering processes with finite state is possible in principle with ensuing latency overhead associated with communication.

- Bare-metal implementation of processes is considered. This is to say, implementation framework for real-time or general purpose operating systems (OSes) execution environments will not be considered.

- Dynamic relocation can occur only when the execution of periodic processes finishes and before it starts again. This means, process relocation does not arbitrary happen at any moment in time.

- Processes which are considered for relocation do exist in all possible nodes to allow low-latency relocation. Otherwise, in principle, process relocation with binary transfer and state can also be handled with ensuing excessive latency overhead associated with binary transfer between homogeneous cores.

- For processing nodes that support mixed-critical execution of processes, a virtualisation solution is assumed. This is to say, the implementation and elucidation of the virtualisation support falls outside the scope of this work.

- A predictable chip-to-chip communication with guaranteed quality of services is assumed to be available.
1.5 Structure of this Document

Chapter 1 introduces the work, describes the problem and its context, and explains the goals and limitation of the work. Chapter 2 provides a brief account on notable historical background and a summary of the related work. Subsequently, Chapter 3 provides the specific design and implementation framework, describes the rationale behind particular design choices and decisions, and briefly provides assumptions related to the target platform and design automation tools used. The proposed design is further analysed and the implications of particular considerations and concerns on various design aspects are discussed in Chapter 4. Lastly, Chapter 5 offers some concluding remarks and suggests future directions.
Chapter 2

Background

2.1 Mixed-Criticality Systems

Mixed-criticality systems arise from the integration desire of processes with different criticality level to the same shared processing platform in various application areas such as in automotive and aerospace industries [22, 23, 24, 25, 26].

In such applications, traditionally hundreds of different processes are mapped to distinct processing or execution units. With the technological advances in semiconductor manufacturing, it is possible to map these processes onto multi-processor system-on-chip, thereby, fewer number of processing units would suffice. Consequently, more efficient utilization of processing resources, reduced physical space and reduced off-chip communication and cabling can be realised potentially reducing overall weight and fuel consumption. Due to the promising benefits of mixed-criticality systems, in future platforms more integration is desired by further exploiting multi-processor system-on-chip platforms. However, this introduced design challenges such as:

- what design methods are needed to manage platform vulnerabilities to interference such that safety and security requirements are met in the presence of processes with different levels of criticality?

- what design methods are required to achieve optimal energy efficiency in such platforms that operate in open and changeable environments?

One way to reduce interferences between processes of different criticality level is by spatially and/or temporally isolating processes running on the same chip. This design mechanism is referred to as partitioning [4]. To enable efficient and effective design methods, several hardware
architectures and execution environments such as specialized operating systems or virtualization environment in multi-processor chips have been proposed and discussed thoroughly in [27, 28, 29]. Additionally, specialised mixed-criticality support by partitioning mechanism in operating systems such as PikeOS, VxWorks, LynxOS and XTRATUM have been presented. The scheduling of multi-critical processes onto multi-processor systems has also been considered a design concern and has received extensive research efforts for example in [30, 31, 32, 33, 34, 35, 36, 37, 38]. Moreover, hardware architectures proposed in literature have heavily exploited the computational intensity of multi-processor system-on-chip and often considered a network-on-chip interconnection methodology due to its scalability, modularity and predictability which will be covered further in more details in this chapter.

2.2 Multi-Processor System-on-Chips

The introduction of monolithic microprocessors into electronic systems added to electronic systems some sort of computation capacity at a low cost, low area and lower power consumption [39]. Since then, processor design observed more development over time due to advances of manufacturing technology as captured by Moore’s law [40]. Example of such developments in late 70s is the introduction of Complex Instruction Set Computers [41] and in the early 80s is the Reduced-Instruction Set Computers [42, 43]. Due to the small code size, lower silicon area utilization and power consumption, processors such as ARM processors [44] were considered to be suitable for special purpose systems commonly known as embedded systems. Single-core RISC processors prevailed till when multi-cores were introduced to provide increased performance due to limitations on frequency scaling.

The introduction of reconfigurable devices, later known as Field-Programmable Gate Array (FPGAs), by exploiting the programmability of volatile memories allowed another degree of adaptability to electronic system development which opened new opportunities for reconfigurable computation. An example for the latter in the context of processor design, is the dynamic instruction set computer [45] as opposed to traditional instruction set [46].

FPGAs are programmable logic devices that use on-chip programmable memory device to store the logic functionality of an implemented design. Several architectures and manufacturers exist but generally share this concept such as [47, 48]. Field programmability feature of FPGAs was an enabler for run-time reconfigurability whose use has been demonstrated in [49].
The advances in Very Large Scale Integration (VLSI) technology allowed more integration of analog circuits, memories, mixed-mode circuits, processors and programmable logic in a single-chip which enabled further applications [50, 51]. A current version that embeds multi-core ARM processor and latest FPGA series in 28 nanometer technology is Xilinx Zynq™-All-Programmable System-on-Chip [5, 52] which is portrayed in Fig. 2.1.

Figure 2.1: Xilinx Zynq-7000: A block diagram depicting typical multiprocessor system-on-chip. The diagram depicts different parts of the system mainly: the processing system and the programmable logic. [52].

2.3 Network-on-Chip Architecture

Network-on-Chip or on-chip networks is considered a scalable interprocessor interconnection architecture for many-core applications providing high communication bandwidth, reduced latency and improved energy-efficiency compared to other alternatives [9, 53, 54, 55, 56]. The major elements of such architecture are switches connected in a particular topology and interfaced to processing nodes through network interfaces (NI). A typical example of a NoC can be seen in Fig. 2.2. The figure depicts four processing resources or
nodes marked as Rxx. The processing resources are connected via network interfaces (NIs) to an array of switches marked as Sxx. There are several terminologies concerning the NoC and to reduce possible ambiguity in the remainder of this work, some terminologies have been defined briefly as follows:

- **Network-on-Chip/On Chip Network**: The collection of switches comprising the network.
- **Switch/Router**: an intermediate module connecting to a processing resource and other switches to relay messages between resources or nodes.

Figure 2.2: Architecture of network-on-chip depicting processing resources, network interfaces and switch network for a 2x2 mesh topology
• **Resource/Node:** one or more processing modules that share a memory address space. The node accesses the network through a network interface to communicate with one or more other network nodes. Usually it is a processor system with its own private local memory and peripherals but, in principle, it could be any type of resource such as memory or special purpose intellectual property (IP) block.

• **Network Interface (NI):** an intermediate module connecting a resource to the network.

• **Packet:** the message in one or more words being sent or received in the network. Usually it consists of a number of flits.

• **Flit:** originally short for flow control digit and is used interchangeably in this work with Phit, physical control digit. A flit is the actual sort of bits propagating from or to the NI through the switches. Flits can be carrying setup information, data words or configuration commands.

• **Routing scheme:** algorithm used to decide the path of a flit propagation through the network, for example, YX-routing scheme in the Nostrum NoC routes a flit horizontally until it reaches its Y position and then vertically until it reaches its X position.

• **NoC Topology:** The connection between the switches, usually in shape of mesh, torus, star, tree, etc.

• **NoC Flow Control:** the procedure employed for transmission and reception of packets to allow correct message delivery between NI and switch and between switches. Examples for flow control include: buffer-less flow control, store and forward flow control, wormhole, etc.

Several switch topologies exist such as mesh, star, tree, torus and cone. The switches are responsible for routing the message also known as a packet. The packet can be sent at smaller chunks called FLow control digITs (FLIT)s. To transport flits from one node to another error handling and flow control mechanisms, and routing algorithms are typically considered [57, 58]. Each configuration choice has an implication on performance, reliability, throughput, bandwidth, energy per transaction, area etc., which are thoroughly documented in the literature such as in [59, 60, 61, 56].
2.4 System Design Automation

To shorten development time, system design automation can be leveraged to design cyber-physical embedded systems [62]. This is due to the potential of making several algorithmically optimal and correct design steps of systems run in a repeatable way at speed of computers rather than humans. Moreover, it helps coping with design complexity by taking away most low-level details and mitigating human errors in the process. As a result, less higher-level parameters exposed to designers to tweak around could enable more complex applications to be done with less resources and possibly give more room for innovation. In principle, systems can be modelled, i.e. using defined models for functionality, process communication and constraints at high level of abstraction. Then semantic preserving transformational algorithms can be applied on the models and constraints to derive suitable implementation at the low level. Such approach can also be called system level synthesis which comprises several classes of design automation such as high-level synthesis. Depending on the hardware primitives used, architectures considered, software execution environment, and incorporated algorithms for design automation, a correct by construction implementation may be achieved.

System design automation has observed tremendous development over the last decades. For instance, according to Grant Martin and Gary Smith, high-level synthesis has gone through evolutionary history since the 1980s to today [63]. This evolutionary process gave rise to high-end, state-of-art commercial tools such as Xilinx Vivado tools [64] that can implement a subset of defined C/SystemC/C++ code into FPGA platform or as a software on an embedded processor. At the expense of losing, sometimes negligible, performance efficiency, such tools provide substantial increase in productivity. However, they still lack further higher level of process modelling such as models-of-computation in [65], to allow formal construction of the systems. In that aspect, the work in [20, 66, 21] can be considered a research direction towards making system design tools defined and modelled at high level, for paving the way towards more design productivity for embedded systems.

Likewise, Hanson et al proposed the CoMPSoC design paradigm, which is a design flow with emphasis on composability, scalability and predictability for real-time embedded systems by employing the AEtherial [59] time-triggered [67] network-on-chip based platform [68]. Several works such as [69, 13] have been done to add considerations for particular design issues such as power management and front-end model-of-computation to the ecosystem of the design flow.
For mixed-critical systems, Obermaisser et al in [28] proposed a multiprocessor system architecture for mixed-criticality systems. Among the features of the proposed system is the exploitation of time-triggered network-on-chip to allow reduced jitter in network access and latency during inter-process communication. Consideration for optional execution environment and middleware layers to support services such as automotive open system architecture (AUTOSAR) is shown. A more concrete NoC based platform suitable for mixed-criticality systems, is the IDAMC introduced in [70]. In this work, more emphasis on dynamic flexibility while preserving isolation requirement is presented. It works on the premise that processes of different criticality get assigned to different priority which allows them to access the network of shared resources instead of employing static time-division multiplexing.

Based on the afore-mentioned pile of research work, the remainder of this work will attempt to answer the following question:

What design framework is applicable for mixed-critical multiprocessor systems to support deterministic dynamic process relocation?
Chapter 3

Design and Implementation

3.1 Overview

The proposed implementation framework for dynamic process relocation (DPR) relies on a simple design concept. The design accept inputs that comply with DPR design constraints to produce the system with DPR capabilities.

The inputs of the design are:

- application sources and their mapping onto the MPSoC,
- application communication graph,
- criticality levels of the processes,
- list of relocatable processes marked with unique process IDs, their initial location and list all possible locations at run-time, and,
- a DPR table consisting of:
  - the conditions, i.e. event or time instances, at which process relocation should occur,
  - the ID of the process being relocated,
  - the source node and
  - the destination node

The design inputs should comply with the following constraints:

- processes mapped onto the same processing units are not allowed to have levels of criticality exceeding that of the processing node.
For instance, processes with two levels of criticality are not allowed to be mapped onto processing nodes that support only one level of criticality. The mixed criticality support of a processor is a feature of the processor architecture that enables the protection of memory spaces and peripherals that belong to higher levels of criticality from being accessed or manipulated by processes of lower levels of criticality.

- The utilisation time of the worst case running processes on any processing core, $U_{wc}$ should not be more than $69\% -$ the DPR overhead utilisation, $U_{DPR}$, i.e.

$$U_{wc} \leq 69\% - U_{DPR}$$  \hspace{1cm} (3.1)

This is because, Liu and Layland criteria for rate-monotonic scheduling is assumed to set a pessimistic lower bound for processes schedulability [71]. However, this constraint can be overlooked, if a scheduler can guarantee the worst case schedulability of processes including the handling processes for DPR.

- The memory capacity at each processing node is more than or equal to the sum of the memory utilisation of the worst case run time processes on a processing core, $M_{wc}$, and that attributed to DPR overhead, $M_{DPR}$. This include both data and instruction memory.

- The inter-processor communication bandwidth available at each processing node is more than or equal to the sum of the communication bandwidth required at the worst case run time processes on a processing core, $C_{wc}$, and that attributed to DPR overhead, $C_{DPR}$.

Here $U_{DPR}$, $M_{DPR}$ and $C_{DPR}$ are design constant dependent on the target platform.

For a set of inputs that comply with the constraints mentioned, the implementation framework produces a system with DPR capabilities. The output system includes, in addition to the processes, other process primitives called process handlers (PH), network daemon (ND) and network proxy (NP). These software primitives are part of proposed extensions referred to in this documents as design framework primitive.

A graphical illustration of the proposed extensions is shown on a two dimensional network with a mesh switching network topology in Fig. 3.1.

The philosophy behind the extensions is to add software primitives that communicate and control the interface controller peripheral or NI to the local interconnection network. As a result, the concept of a master node, network daemon and processes handlers are introduced. The extension brings the following features:
3.1. Overview

![Diagram](image)

**Figure 3.1:** A diagram depicting process handlers and network daemon on a 2x2 NoC-based MPSoC
1. Control on when other processes should start and suspend.
2. Control on process relocation.
3. Providing a gateway for packets intended to be sent off network.

This is enabled by the following:

1. The marking of the NI connected to the master node as the master NI such that special control for process start time synchronisation, initiating process relocation, and access to off network gateways or channels can be issued and controlled.

2. The addition of software services to the network driver at the master and non-master nodes. The collection of those services at the master node is called the local network daemon, whereas at the non-master nodes are called process handler.

3. The physical enabling of all possible communication channels in the switch network hardware is done at design time, while leaving the control in run-time to the network daemon and process handlers.

In order for the DPR implementation to work correctly, additional meta-data should be provided to the process handlers and network daemon. For example, at each processing node with processes subject to dynamic relocation, the following information has to be produced and provided during the design implementation:

1. Process equivalence tables.
3. DPR communication forwarding decision trees.

Furthermore, to avoid ambiguous roles and responses a DPR protocol and DPR instruction sets are considered.

A summary for the design and implementation flow mentioned previously is given in Fig. 3.2. The figure shows the inputs required for the design and the final outputs products of the implementation. In order to put the produced outputs into an implementation perspective, an illustration for the components in a possible system implementation in context of DPR is given in Fig. 3.3. This figure shows an example for communicating processes with mixed criticality (blue to indicate low critical process and red to indicate high critical process) mapped onto NoC based MPSoC. The figure also shows
3.2. Design Assumptions

The target platform considered for this implementation framework is an instance of 2x2 Nostrum-based network-on-chip for on-chip communication developed with the aid of a network-on-chip system generator (NSG) [72]. The architecture and principle of operation is given in [19, 72]. The design blocks used and the synthesis and characterisation results are given in the Appendicies.

Network-to-network bridges including chip-to-chip communication and inter-operating system is done as part of another tool-chain by Youssef Zaki [73]. In essence, the other work, is built around the idea of embedded local-cloud in which the messages can be conveyed to the processes regardless of its location. The work mainly uses a propriety solution called SHAPE developed primarily on Linux and is deployable in a networked distributed environment. The concept of the work can be seen in [74]. Virtualisation techniques were used leveraging ARM-trustZone technology to compose a mixed-criticality within the dual ARM Cortex A9 processing subsystem.
Figure 3.3: Dynamic process relocation (DPR) scenarios and relation with process handlers, network daemon and network proxies. Four relocation scenarios are presented. The relocatable processes are marked with asterisks (*).
The rest of the this chapter describes the design and the implementation flow in more details.

3.3 Design Framework Primitives

3.3.1 Master NI

This master NI is connected to the node which has the capability to relay off-network packages, and has the control on process synchronization and process relocation. The synchronization and relocation is done by a permanent process at the master node called network daemon providing two main functionalities: network synchronisation and configuration control. Illustrative flowchart for the background service handler of the network daemon is given in Fig. 3.4.

The master node could be any node in principle. For a MPSoC such as the Zynq, there are advantages in using the Dual ARM as the master node. First, it is a hard-wired IP implying high energy efficiency operation when controlling memory controllers and off-chip communication peripherals such as CAN and Ethernet. Second, Zynq devices are configured such that the ARM multicore is the first to initialise and it has a control on the configuration of the FPGA fabric.

In the ARM node there exist two regions for the critical and non-critical processes implementing hyper-virtualisation technique exploiting ARM TrustZone™ hardware feature. This configuration effectively allows treating the two zones as isolated, meaning non-critical processes cannot interfere with the inter-process communication or memory space of critical processes and vice versa. There are two channels for network-to-network communication, one for critical processes and the other for non-critical processes. At the critical section resides the network daemon, which controls processes at the network and relays its off network packets. On the non-critical process region, a network proxy (NP) is present. The functionality of a NP is to handle communication between different criticality regions within the same node and also handle communication to off-network nodes. The network proxy service is there to abstract the off network gateway communication and therefore is used to relay off network messages. The communication between the network daemon and network proxy service occurs via a shared-memory region detailed as in Fig. 3.5. The figure shows for example if a network daemon wants to relay a message to a critical process, \( P_i \), located at chip \( x \), the network daemon will write packets to the respective channel in the shared memory.
Additionally, the principle allows the master node to control global timing within the scope of the network and affect the routing algorithm for switches. It can also be considered as the node where fault detection, containment and recovery commands are issued, power states transition control is carried out, power supply monitoring and management is taking place, within-die temperature monitoring and thermal energy dissipation control services are handled.

A set with limited instructions (commands) is used to allow intelligible communication between the network daemon and the process handlers. The commands considered for this work are as in Fig. 3.6 and command bit fields are shown in Fig. 3.7.
3.3. Design Framework Primitives

Figure 3.4: Network daemon service at the master node
### Command channel

<table>
<thead>
<tr>
<th>Critical processes inbox channels</th>
<th>Network ID</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical processes outbox channels</td>
<td>Network ID</td>
<td>Packets</td>
</tr>
<tr>
<td>Non-critical processes inbox channels</td>
<td>Network ID</td>
<td>Packets</td>
</tr>
<tr>
<td>Non-critical processes outbox channels</td>
<td>Network ID</td>
<td>Packets</td>
</tr>
</tbody>
</table>

**Figure 3.5:** Shared memory region where network daemon and network proxies can access to receive and relay off-network communication

<table>
<thead>
<tr>
<th>Command</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start network processes</td>
<td>sttn</td>
</tr>
<tr>
<td>Kill network processes</td>
<td>stpn</td>
</tr>
<tr>
<td>Start Non-Critical Process</td>
<td>kill p</td>
</tr>
<tr>
<td>Stop Non-Critical Process</td>
<td>stt p</td>
</tr>
<tr>
<td>Relocate Process</td>
<td>mov p, src, dest</td>
</tr>
<tr>
<td>Alive Signal</td>
<td>alive p</td>
</tr>
<tr>
<td>Process State Transfer Begins</td>
<td>ststtb</td>
</tr>
<tr>
<td>Process State Transfer Ends</td>
<td>ststte</td>
</tr>
</tbody>
</table>

**Figure 3.6:** List of instructions and corresponding mnemonics used for the communication between network daemons and Process Handlers. P denotes process ID, src denotes the source node at which the process resides and dest denotes the destination node at which the process would eventually reside.

<table>
<thead>
<tr>
<th>opcode</th>
<th>pid</th>
<th>src</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bits</td>
<td>4 bits</td>
<td>10 bits</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

**Figure 3.7:** Instruction/command format: opcode field corresponds to a unique numeric representation of the instructions considered in the DRP instruction set
3.3.2 Process Handler

The process handler is a permanent service at all non-master nodes that keeps listening for instructions from the network daemon and act accordingly to start processes at power-on and activate and de-activate processes dynamically as demanded by the network daemon.

![Process handler flowchart](Figure 3.8: Process handler at non-master nodes)
3.3.3 Network Channels Allocation

Network channels are virtual channels that effectively occupy a memory address range for inbox and outbox messages. All process to process communications are assigned unique virtual channels. At the sending process side, the virtual channel is called send channel and at the receiver side it is called receive channel. In DPR context, virtual channels are allocated for all possible locations of processes in addition to virtual channels between network daemon and process handlers. The purpose of the channel allocation step is to ensure that at all cases, there exists a defined memory space recognisable by the NI to deposit and fetch packets for all possible process relocation scenarios. Allocating channels between network daemon and process handlers allow process synchronization and commanding of process relocation, acknowledging relocation activities, transfer of process state, message forwarding and process health monitoring.

To implement DPR correctly, channel allocation and positioning of processes in the routing tables have to be addressed. For that matter, channel allocation for all possible process relocation strategies are made at design time to allow predictability analysis on message delivery for all possibilities of relocation. A set of rules has been defined to make it possible to manage process relocation in a predictable manner. The DPR rules are as follows:

1. There should be one process handler per node except for the master node. The master node has a network daemon.

2. The network daemon has bi-directional channels to all process handlers to enable the flow of commands and other control signals.

3. Applications processes can be created and mapped to nodes. Depending on whether they are permanent or not, their inter-process communication can go through process handlers or can be direct.

   (a) If the source and destination processes are permanent, they communicate directly.

   (b) A relocatable process which neither has source nor destination process is categorised as type 0. Such a process has implications on execution but none with regards to message forwarding. (see relocation scenario type 0 in Fig. 3.3).

   (c) If the source process is possible to be relocated within the network while the destination process is not, the source should communicate to the process handler at the node which relays the
message to the destination process at that node. This can be categorised as relocation scenario type 1.

(d) If the source process is permanent while the destination process is possible to be relocated within the network, the source should communicate to all process handlers at the nodes which in turn relay the message to all possible destination processes. This can be categorised as relocation scenario type 2.

(e) If the source process is possible to be relocated outside the network while the destination process is not, the source should communicate to the network daemon which tracks the source process and relays the message to the destination process at that node.

(f) If the source process is permanent while it is possible to relocate the destination process outside the network, the source should communicate to the network daemon, which in turn identifies the location of the destination process and relays the message either off the network or on the network accordingly. The relocatable process outside the network is classified as type 3.

Fig. 3.9 shows an overview of the design framework primitives to support DPR. This includes the network daemon at the master node, process handlers at non-master nodes with processes subject to relocation, all existing in a local network. A network to network bridge is used to connect multiple local networks. A network proxy is used to facilitates the communication between different networks.
Network Interface (NI) memory map

<table>
<thead>
<tr>
<th>Channels</th>
<th>Control Processes</th>
<th>Packets</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.9: Diagram depicting the general theme of the implementation and the important concepts. Nodes can be single or multi level of criticality. All NIs have special isolation in memory for inbox and outbox per process. Master NI has special commands for synchronisation and process control. Each node has either network daemon, Network Interface (NI) memory map, or Process Handlers (PH) assigned high criticality level and permanent dormancy and have control on process execution states and channels.

**ND** (Network Daemon) **PH** (Process Handler) **NP** (Network Proxy)

ND: Synchronises intra-node processes/controls intra-node forwarding channels

PH: Synchronises intra-node processes/controls intra-node forwarding channels

NP: Relays out-of-network messages

Nodes can be single or multi level of criticality. All NIs have special isolation in memory for inbox and outbox per process. Master NI has special commands for synchronisation and process control. Each node has either network daemon, Network Interface (NI) memory map, or Process Handlers (PH) assigned high criticality level and permanent dormancy and have control on process execution states and channels.
3.3.4 Dynamic Process Relocation Protocol

The proposed process relocation protocol is given in Fig. 3.10.

Master Node:
Init. and get proc. state from source node

Source Node:
- Save proc. state
- Send proc. state

Master Node:
- Send P state to dest. node

Dest. Node:
- If state received, Send ACK

Master Node:
- De-act P at Source Node
- Send Act P at Dest. Node

Continue if ACK received, Cancel DPR if timed-out

Figure 3.10: State diagram depicting process relocation protocol

This protocol dictates the actions, responses, steps and conditions that network daemons at master nodes and process handlers at the source nodes and destination nodes shall comply with to achieve correct DPR. The process starts by the network daemon (ND) at the master nodes. The ND initiates the process by sending DRP instruction with arguments containing the process to be relocated to the process handler (PH) at the source node. This in effect translates into requesting the execution state of process ID to be relocated (process context). The PH at the source node responds by sending the state of the process to be relocated to the ND. The ND in turn send the PH at the destination node the state of the process. Upon receiving an acknowledge from the PH at the destination node the ND sends two commands to the PHs at the source and destination nodes to stop and activate the process at the source and destination nodes respectively. The instruction sets described in Fig. 3.6.
3.4 Implementation

3.4.1 Potential Scenarios

Several implementation scenarios can be considered for DPR including:

1. Intra-network dynamic process relocation for processes acting as source process. For example, a process that reads a redundant input. A redundant input is an additional replica of an input module that is wired to one or more systems to provide additional input spare for improved reliability and flexibility.

2. Intra-network dynamic process relocation for processes acting as a destination process. For example, a process that controls a redundant hardware output such as a valve or fuel injector. A redundant output is an additional replica of an output module that is wired to one or more systems to provide additional output spare for improved reliability and flexibility.

3. Inter-network dynamic process relocation for processes acting as source process. For example, a process that reads a sensor such as airflow sensor connected over multiple chips.

4. Inter-network dynamic process relocation for processes acting as a destination process.

In the implementation, the following four scenarios are considered:

1. Intra-chip relocation of a critical process connected with input acting as source process while preserving the position of the destination process.

2. Intra-chip relocation of a critical process connected with an output acting as a destination process while preserving the position of the source process.

3. Inter-chip relocation of a critical process that does not communicate with any process.

4. Inter-chip relocation of a non-critical process connected with two other processes, one of them is a source connected to an input and the other is a destination connected to an output. Both source and destination processes are fixed while the middle process can relocate from the Microblaze on the FPGA to the non-critical section of the ARM to another ARM off chip to another FPGA area off network.
Case 1: Input 1: \( p_1^* \rightarrow p_2 \rightarrow \) Output 1

Case 2: Input 2: \( p_3 \rightarrow p_4^* \rightarrow \) Output 2

Case 3: Input 3: \( p_5^* \rightarrow \) Output 3

Case 4: Input 3: \( p_6 \rightarrow p_7^* \rightarrow p_8 \rightarrow \) Output 3

Figure 3.11: Implementation scenarios showing inter-process communication. Processes \( p_1, p_2, \ldots, p_5 \) are coloured in red to denote their high criticality level whereas \( p_6, p_7 \) and \( p_8 \) are coloured in purple to indicate their low criticality level. The asterisk on processes denotes that they are subject to relocation.

The implementation scenarios have been explained in Fig. 3.11. Table 3.1 shows two possible dynamically reconfigurable states that can be switched. The events causing DPR events are simulated by a switch called DPR switch.

Table 3.1: DPR table. The table describes two states that can be dynamically switched depending on events simulated by switches. The default configuration is underlined.

<table>
<thead>
<tr>
<th>Event</th>
<th>DPR Mode Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPR SW=0</td>
<td>Minimal Energy Consumption: Non-critical processes are shut-off in both chips</td>
</tr>
<tr>
<td>DPR SW=1</td>
<td>High Performance/Balanced Load: Load is distributed, all features available, ( p_5^* ) is enabled in chip 2 only</td>
</tr>
</tbody>
</table>

The table can be expanded to enable fault tolerance or cloud-like computation scenario by defining mode in which \( p_5^* \) and \( p_7^* \) can be instantiated over other chips.
3.4.2 Implementation Flow

The implementation comprises the following main steps:

1. The creation of processes corresponding to network daemon (ND) and Process Handlers (PH) at the master and non-master nodes: a network proxy process is made to abstract off-network communication. In this work, off-network (off-chip) communication is handled by SHAPE propriety solution. Fig. 3.12 illustrate this step.

   ![Diagram](image)

   Figure 3.12: Network daemon and process handler channels dedicated for synchronisation, monitoring and process relocation. The orange colour signifies their permanent dormancy and activeness in their respective nodes

2. The rest of the processes are constructed and the communication directions or channels are established in light of the use cases in Section 3.4.1 and rules laid-down in Section 3.3.3. Fig. 3.13 and Fig. 3.14 show the process constructed and channels allocated for two chips.
Figure 3.13: Processes at chip 1. The processes marked with asterisk are relocatable.
Figure 3.14: Processes at chip 2. The processes marked with asterisk are relocatable.
3. After creating the diagrams, then the design can be transferred to the NSG to generate the system with all allocatable channels available. This is shown in Fig. 3.15. Similar work has been applied in all networks in all chips and for brevity, the following steps are shown for the network at chip 1 only.

Figure 3.15: Enumeration of processes and communication channels at chip 1 using in-house network-on-chip system generator. Notice that all processes are populated and the black and white colouring indicates that criticality and default process status is transparent to this step.

4. The generated system is modified to work on the target device as illustrated in Appendix A.3.
5. After hardware implementation is done, channels IDs, process IDs and equivalent processes are identified from the generated C codes by the NSG. This is shown in Table 3.2, Fig. 3.16 and Fig. 3.17.

Table 3.2: List of allocated channels for chip 1

(a) Receive channels

<table>
<thead>
<tr>
<th>Ch.</th>
<th>Node: 0</th>
<th></th>
<th>Ch.</th>
<th>Node: 1</th>
<th></th>
<th>Ch.</th>
<th>Node: 2</th>
<th></th>
<th>Ch.</th>
<th>Node: 3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>src</td>
<td>dst</td>
<td></td>
<td>src</td>
<td>dst</td>
<td></td>
<td>src</td>
<td>dst</td>
<td></td>
<td>src</td>
<td>dst</td>
</tr>
<tr>
<td>0</td>
<td>P10</td>
<td>P0</td>
<td>0</td>
<td>P12</td>
<td>P1</td>
<td>0</td>
<td>P6</td>
<td>P2</td>
<td>0</td>
<td>P0</td>
<td>P3</td>
</tr>
<tr>
<td>1</td>
<td>P9</td>
<td>P0</td>
<td>1</td>
<td>P6</td>
<td>P1</td>
<td>1</td>
<td>P0</td>
<td>P2</td>
<td>1</td>
<td>P0</td>
<td>P0</td>
</tr>
<tr>
<td>2</td>
<td>P3</td>
<td>P0</td>
<td>2</td>
<td>P4</td>
<td>P1</td>
<td>2</td>
<td>P2</td>
<td>P13</td>
<td>2</td>
<td>P0</td>
<td>P11</td>
</tr>
<tr>
<td>3</td>
<td>P2</td>
<td>P0</td>
<td>3</td>
<td>P0</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>P1</td>
<td>P0</td>
<td>4</td>
<td>P1</td>
<td>P5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>P1</td>
<td>P7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Send channels

<table>
<thead>
<tr>
<th>Ch.</th>
<th>Node: 0</th>
<th></th>
<th>Ch.</th>
<th>Node: 1</th>
<th></th>
<th>Ch.</th>
<th>Node: 2</th>
<th></th>
<th>Ch.</th>
<th>Node: 3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>src</td>
<td>dst</td>
<td></td>
<td>src</td>
<td>dst</td>
<td></td>
<td>src</td>
<td>dst</td>
<td></td>
<td>src</td>
<td>dst</td>
</tr>
<tr>
<td>0</td>
<td>P0</td>
<td>P10</td>
<td>0</td>
<td>P1</td>
<td>P7</td>
<td>0</td>
<td>P2</td>
<td>P13</td>
<td>0</td>
<td>P3</td>
<td>P0</td>
</tr>
<tr>
<td>1</td>
<td>P0</td>
<td>P11</td>
<td>1</td>
<td>P1</td>
<td>P5</td>
<td>1</td>
<td>P2</td>
<td>P0</td>
<td>1</td>
<td>P10</td>
<td>P0</td>
</tr>
<tr>
<td>2</td>
<td>P0</td>
<td>P2</td>
<td>2</td>
<td>P1</td>
<td>P0</td>
<td>2</td>
<td>P12</td>
<td>P1</td>
<td>2</td>
<td>P11</td>
<td>P0</td>
</tr>
<tr>
<td>3</td>
<td>P0</td>
<td>P3</td>
<td>3</td>
<td>P4</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>P0</td>
<td>P1</td>
<td>4</td>
<td>P6</td>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>P6</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2 describes how the receive channels Table (a) and send channels in Table (b) are allocated at each node. For every channel, the source and destination process are listed.
### 3.4. Implementation

<table>
<thead>
<tr>
<th>Org Process</th>
<th>Equivalent Enumerated Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>P4  P12</td>
</tr>
<tr>
<td>p4</td>
<td>P7  P13</td>
</tr>
<tr>
<td>p7</td>
<td>P10  *</td>
</tr>
</tbody>
</table>

**Figure 3.16:** Process equivalence table at chip 1. P* denotes a process that is located outside the scope of the network.

Fig. 3.16 shows the list of equivalent enumerated processes for every relocatable process, whereas Fig. 3.17 describes the generated process status table to be available at the respective ND.

<table>
<thead>
<tr>
<th>Chip 1 (ND)</th>
<th>Chip 2 (ND)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PH1</td>
<td>PH1</td>
</tr>
<tr>
<td>PH2</td>
<td>PH2</td>
</tr>
<tr>
<td>PH3</td>
<td>PH3</td>
</tr>
<tr>
<td>Cloud</td>
<td>Cloud</td>
</tr>
</tbody>
</table>

**Figure 3.17:** Process status table
6. The process relocation implication tree per node for all possible cases is constructed. The diagram shows the IDs of processes activation and de-activation in addition to message channel source and destination forwarding. This is shown in Fig. 3.18 and Fig. 3.19.

![Diagram](image)

Figure 3.18: Construction of dynamic process relocation decision trees for packet forwarding. Ch\(^*\) denotes channel for off-network packets.

At this stage, all meta-data required for the implementation of the process handlers is available. The code for process activation and de-
activation and message forwarding at the process handlers and network daemon is generated accordingly.

**DPR Commands**

mov P1,N1,N2: $P_{4s} \rightarrow 0, P_{12s} \rightarrow 1$

mov P1,N2,N1: $P_{4s} \rightarrow 1, P_{12s} \rightarrow 0$

mov P4,N1,N2: $P_{7s} \rightarrow 0, P_{13s} \rightarrow 1$

mov P4,N2,N1: $P_{7s} \rightarrow 1, P_{13s} \rightarrow 0$

mov P7,N3,*: $P_{10s} \rightarrow 0, P_{*s} \rightarrow 1$

mov P7,*N3: $P_{10s} \rightarrow 1, P_{*s} \rightarrow 0$

**Figure 3.19:** Comprehensive dynamic process relocation command effects on packet forwarding decision trees and corresponding dissemination of activation/de-activation commands to respective process handlers. Ch* denotes channel for off-network packets.
Chapter 4

Results and Discussion

This chapter reflects on the quantitative characteristics of the implementation and briefly discusses their implications.

4.1 Results

The implementation has been made as described in the previous chapter for periodic and stateless processes with a period of $1\mu$ Seconds. The design was synthesized at 50MHz on "xc7z020 clg484-1" chips. Reports on exact resource utilisation, timing, energy consumption are extracted from Vivado. Further characterisation is given in Appendix B.

As far as dynamic process relocation (DPR) latency and overhead is concerned, the following results in Table 4.1 summarised the overheads incurred by the DPR support.

Table 4.1: DPR Overhead

<table>
<thead>
<tr>
<th>DPR parameter/metric</th>
<th>Number of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes initialisation latency ($T_{DPR}^{ini}$)</td>
<td>288 ($3 \times T_{packet}$)</td>
</tr>
<tr>
<td>Process status change overhead ($T_{chg}^{s}$)</td>
<td>10</td>
</tr>
<tr>
<td>$3 \times (\max N_{DPR}^{T_i} + 1)$ $\forall T_i \in [0..N]$</td>
<td></td>
</tr>
<tr>
<td>Process status check overhead ($T_{chk}^{s}$)</td>
<td>5</td>
</tr>
<tr>
<td>Packet forwarding latency ($T_{packet}^{forwarding}$)</td>
<td>197 ($2 \times T_{packet} + T_{chk}^{s}$)</td>
</tr>
<tr>
<td>Process relocation latency</td>
<td>202 ($2 \times T_{packet} + T_{chg}^{s}$)</td>
</tr>
</tbody>
</table>

In Table 4.1, the latency and memory (instruction cycles) overhead is expressed in units of cycles. $T_{packet}$ is 96 cycles for a 32-bit word and its calculation is provided in Appendix B.
The processes initialisation latency, denoted by $T_{DPR}^{ini}$, refers to the minimum time required for the network daemon to command process handlers to start the execution of processes. Since in this implementation there are only three process handlers and therefore three start commands are required to be sent serially. Each command is sent over a packet hence $3 \times T_{packet}$ for the minimum latency.

The worst-case process status change overhead, $(T_{chg}^s)$, depends on the maximum number of relocatable process exists in all nodes ($\max N_{DPR}^T \forall T_i \in [0..N]$).

This is because, the larger the number of relocatable processes, the more comparisons are required for knowing to find which process to change. This is in addition to extra instruction required to actually change the status of the right relocatable process. On the other hand, the process status check overhead $(T_{chk}^s)$ depends on the number of instructions required to access the relocatable process status table and the comparison required to check the process status. This is required each time for packet forwarding.

Due to the possibilities of dynamic relocation, packets could be subject to forwarding thus introducing a latency denoted as $T_{packet}$. The new latency due to forwarding effects ($T_{forwarding}$) and the overall latency can be denoted as $T_{packet}$. Since the forwarding is done within the process handlers/network daemon, it is highly dependent on the software implementation of the forwarding. Mainly, the time required to check the status of the destination and source processes, and the time required to point to the respective inbox/outbox channel. The additional latency is dependent on the processor frequency as well. The $T_{forwarding}$ can be described as follows:

$$T_{forwarding} = T_{packet} + T_{forwarding}$$ (4.1)

$T_{forwarding}$ can be characterised at software development time but usually is considerably smaller than $T_{packet}$.

Dynamic process relocation latency has a max latency, $T_{DPR}^{max}$, dependent on the size of the state, process binaries and cycles required to store the context. Dynamic process relocation latency can be described as follows:

$$T_{DPR} = T_{De-act. Comm.} + T_{state} + T_{Act. Comm.} + T_{state} + T_{ack.}$$ (4.2)

Assuming process activation latency is $\ll T_{DPR}$ and $T_{Act. comm.} = T_{ack. packet} = T_{De-act. comm.} = T_{packet}$ and that the state transfer latency $T_{state}$, is:

$$T_{state} = T_{packet} \times \left\lceil \frac{\text{Size of State}}{\text{Size of packet}} \right\rceil$$ (4.3)

$$T_{DPR} \approx 3T_{packet} + 2T_{packet} \times \text{Size of State}$$ (4.4)
4.2 Discussion

When the protocol does not employ operation completion acknowledgement and the process is simply restarted without state transfer or when the process is stateless:

\[ T_{DPR} \approx 2T_{packet} \] (4.5)

From this characterisation more interesting analysis can be made. For instance, the worst-case overall DPR overhead can be calculated depending on worst-case number of packets to be forwarded or redirected and the worst-case number of processes to be activated/de-activated at a time. In this implementation, the worst-case can be found in node 2 (see row 2 and column 1 of Table 3.17).

Furthermore, the framework has been demonstrated for mesh network-on-chip based multiprocessor systems. However, it could be applied to point-to-point interconnection network or bus-based shared-memory multiprocessor system with different temporal characterisation especially \( T_{packet} \).

4.2 Discussion

The proposed process relocation methodology provides some possibilities at some costs. This sets a design problem that needs to be addressed to achieve optimal implementation for the use-case under consideration. For instance, among the design overheads is: the information redundancy for the replica processes and computational and networking overhead of the process relocation management. In order to reduce the static process overhead in memory, it is possible to send the process binaries to be executed before activating the process. This could reduce the memory overhead but would add networking traffics and elongation of relocation latency in addition to complexity in handling of binary compatibilities within heterogeneous MPSoCs. This trade-off has to be taken into account during the design process.

Additionally, the processing nodes considered here do not take into account hardware reconfigurability. Hardware reconfiguration can be made as a prior conditioning to the node before the actual process being relocated. This also makes the relocation longer but has the advantage of adding another degree of freedom on how process migration can be implemented. In the rest of this section, the effect of dynamic process relocation on energy consumption and reliability is briefly highlighted. A few remarks on design optimisation for relocation are subsequently made.
4.2.1 Potential Energy Consumption Reduction

Due to the dynamic process relocation capability, energy consumption improvement is expected with the assumption that processors are only active when they have processing load to do. This can be realised by setting processor to sleep or scale down the frequency such that the processor is fully utilised. With this assumption, the energy consumption can be related to the execution profile of processes as a function of time. As a result, reduction in energy consumption can be achieved when there are less active processes. Consequently, an improvement of energy reduction can be described as follows:

\[
\text{Energy improvement} = \frac{\sum_{p=0}^{M} \left( \int_{t_0}^{\infty} P_{\mu P}(p,t) dt \right)}{\sum_{p=0}^{N} \left( \int_{t_0}^{\infty} F_A(p,t) P_{\mu P}(p,t) dt \right)} \quad (4.6)
\]

Here:

- \( N \) and \( M \) are the total number of processes on the system when all processes are active versus when dynamic process relocation approach is used.
- \( F_A(p,t) \) is the process activation function profile versus time.
- \( P_{\mu P}(p,t) \) is the instantaneous power consumption versus time, \( t \), of the processing node including process communication at which the process \( p \) resides.

In process relocation scenario, \( F_A(p,t) \) can be zero for finite time intervals which saves more energy compared to situation in which process are active always.

Process load variation as a result of dynamic process relocation can be exploited by making dynamic hardware reconfiguration of the processing node such that minimum allocation of hardware resources for the processing load demand is made.

4.2.2 Potential Reliability Improvement

Making it possible to relocate processes, potentially improves that specific relocatable process reliability as follows:

\[
R_{\text{improved}} = 1 - \left( \prod_{n=1}^{N} \left( 1 - R_{\text{original}} \right) \right) \quad (4.7)
\]
4.2. Discussion

Whereby $N$ denotes the possible number of relocatable positions within the system.

This, however, does not take into account the reduced reliability introduced by the increased design complexity of the network. This is because, the more complex a system gets, the more susceptibility to errors, thus less reliability it gets.

$$R_{\text{system}}^{\text{original}} = \left( \prod_{P=1}^{P_T} R_P^{P_{\text{original}}} \right) \times R_{\text{network}}^{\text{original}}$$  \hspace{1cm} (4.8)

$$R_{\text{system}}^{\text{improved}} = \prod_{P=1}^{P_T} \left( 1 - \left( \prod_{n=1}^{N} \left( 1 - R_P^{P_{\text{original}}} \right) \right) \right) \times R_{\text{network}}^{\text{DPR}}$$ \hspace{1cm} (4.9)

In Equations 4.8 and 4.9, $R_{\text{network}}^{\text{original}}$ and $R_{\text{network}}^{\text{DPR}}$ denote the reliability of the network without any dynamic process relocation and the reliability of the network with dynamic process relocation. Consequently from reliability perspective, the equations may raise an interesting reliability question as to how much process relocation possibilities, $N$, per process would justify the added design complexity of the network. The answer of which would rely on the assumption of the source of failure, process relocation overhead and network size which is application and implementation dependent.

Therefore, it could be stated that, this work allows potential reliability and availability improvement by means of process relocation. However, an optimal mechanism or strategy for detecting the need for process relocation and determining which specific process needs to be relocated is dependent on the design.

4.2.3 Design Optimisation

The use of the in-house, NoC system generator allows to make several customisation with regards to packet size, resource to network bus-width interface and network topology. This allows network designers to tailor the design in light of the actual inter-process communication scenario to optimise message passing latency and energy per message. For example, in a network with larger number of nodes, packet propagation time through the network can increase in a manner dependent on how the nodes are placed, routing algorithm, network topology and flow control.

Additional prospective customisation which this work does not consider is the case in which network switches are susceptible to faults. Such consideration would impose routing procedure to be dynamic such that switch faults can be detected and/or corrected in case of transient faults.
For permanent faults flits re-routing can also be considered. With the presence of faults the temporal characteristics of the communication acquires a probabilistic aspects to it and yields it less deterministic.
Chapter 5

Concluding Remarks

The use of network-on-chip in a multi-processor system-on-chip, which has hard processor and a programmable logic to act as glue logic or hardware acceleration, is advantageous in mixed-criticality system due to the potentiality of scalable design that guarantees interference-free communication between critical and non-critical processes residing on the same FPGA. Integrating processes from multiple chips each using single core processor to a single multi-processor system on chip increases thermal intensity. As a result, dynamic process relocation for load balancing or fault mitigation could be required. Additionally, it would allow, mutually-exclusive and computationally-intensive processes to be executed on the same processing node hence demonstrating efficient resource utilisation. From energy perspective, the network on chip would be a good option when processing nodes exceed a particular threshold depending on technology nodes and number of bus master and access pattern to shared slave resources (e.g. peripherals and memory). Additionally, networks-on-chip could provide channel isolation for inter-processes communication of processes with different levels of criticality.

In this work, an implementation for mixed-criticality network-on-chip suitable to support low-jitter dynamic process relocation has been presented. It provides potential improved increased availability and reliability compared to static configuration with predictable reconfiguration overhead. This can be considered a step towards mixed hardware and software based reconfiguration for improved energy efficiency and more design trade-off freedom for system level synthesis of cyber-physical systems from specifications to implementation.
5.1 Contributions

The work presented here demonstrates design and implementation framework that allows predictable and reliable support for dynamic process relocation in multiprocessor system-on-chip. It introduces design primitives such as: process handlers, network daemon and network proxies in addition to the notion of master and non-master nodes and the dynamic process relocation protocol. The framework may provide designers with additional degree of freedom to meet efficient dynamic adaptability requirements. This is advantageous for load-balancing and fault tolerance which could eventually lead to increased system efficiency, reliability and availability.

5.2 Future directions

System level synthesis from high-level of abstractions is a favourable way of managing design complexity and increasing opportunities for innovation in future development tools. In order to propel development wheel in the system design automation of cyber-physical embedded systems, the following are deemed appropriate future directions:

- Moving fixed algorithms used in the software driver of the network interface to hardware to improve energy efficiency and reduce relocation latency.

- Incorporating dynamic process relocation services within compilation and execution environments, and real-time operating system.

- Employing hardware reconfiguration in addition to the software based process relocation to leverage the potential energy efficiency.

- Development of system design automation and automatic design space exploration to help in fast and optimal system design.
Bibliography


network using FPGAs and run-time reconfiguration,” in IEEE Workshop
1994, pp. 180–188.

Journal, The Authorative Journal for Programmable Logic Users,


[53] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Öberg,
K. Tiensyrja, and A. Hemani, “A network on chip architecture and
design methodology,” in IEEE Computer Society Annual Symposium on

[54] L. Benini and G. De Micheli, “Networks on chips: a new SoC paradigm,”

a noc and a traditional interconnect fabric with layout awareness,” in
Proceedings of the Design Automation Test in Europe Conference,
vol. 1, March 2006, pp. 1–6.

[56] S. Tota, M. Casu, M. Ruo Roch, L. Macchiarulo, and M. Zamboni, “A
Case Study on NoC-Based Homogeneous MPSoC Architectures,” IEEE
Transactions on Very Large Scale Integration (VLSI) Systems, vol. 17,

[57] K. Tatas, K. Siozios, D. Soudris, and A. Jantsch, Designing 2D and 3D
Network-on-Chip Architectures. New York: Springer Science+Business
Media, 2014.

[58] M. Palesi and M. Daneshtalab, Eds., Routing Algorithms in Networks-

[59] K. Goossens, J. Dielissen, and A. Radulescu, “AEthereal Network on
Chip: Concepts, Architectures, and Implementations,” IEEE Design


Appendices
Appendix A

Intellectual Property Blocks

A.1 Embedded Processor Nodes

The embedded processing nodes were the Dual Cortex A9 processor [16, 15] and Microblaze softcore processor [17]. A design flow adopted from [75] has been followed to instantiate and configure the processors accordingly. In favour of predictability and low resource utilization of Microblaze systems, the cache and error correction features have not been used.

A.2 Network-on-Chip IP Blocks

The NoC comprises two IP blocks: a network of switches called the NoC2x2 and the interface to the NoC switches which is called Network Interface, NI, which are described as follows.

The NI has two interfaces, direct access port (DAP) whose data bus width is 32 or 64 bit and AXI bus compatible with ARM AMBA bus specifications [76]. The DAP interface is intended to connect to resources such as a custom Very Large Instruction Word (VLIW) processor, whereas the AXI interface is compatible with ARM Application Processors and Xilinx Microblaze Microprocessor.

The NoC2x2 IP is a network of 4 switches each is connected to a resource through (54 bits bi-directional links and read/write/sync single-bit signals). The sync signal, otherwise called heartbeat signal, allows synchronous scheduling of processes and message passing. The configuration parameters for the NI and the NoC2x2 IPs are shown in Table A.1 and Table A.2 respectively.

The entire system however makes use of several other IPs which are listed in Table A.3.
### Table A.1: Network interface (NI) configuration

<table>
<thead>
<tr>
<th>Configuration Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NI Number</td>
<td>An identifier for the network interface. Each resource, usually a processor, should have a unique identifier for the interface to the network.</td>
</tr>
<tr>
<td>Base Address</td>
<td>The first accessible memory location within the NI, defaulted to 0x7002000</td>
</tr>
<tr>
<td>High Address</td>
<td>The last accessible memory location within the NI, defaulted to 0x7003FFFF</td>
</tr>
<tr>
<td>Address Range</td>
<td>High Address - Base Address + 1: 128KB</td>
</tr>
<tr>
<td>USE 64bit</td>
<td>Boolean number to indicate whether 64bits or 32bits Direct Access Port (DAP) will be used or not. DAP is intended to be used with custom Very Large Instruction Word (VLIW) microprocessor-kind of resource.</td>
</tr>
</tbody>
</table>

### Table A.2: Switches network configuration

<table>
<thead>
<tr>
<th>Configuration Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeartBeat Constant</td>
<td>A constant to set a periodic one clock-cycle signal, HeartBeat or Sync signal, every number of clocks (e.g. 5000000 corresponding to 1 Hz and 500 corresponding 10KHz for a system clock frequency of 50MHz).</td>
</tr>
</tbody>
</table>
A.2. Network-on-Chip IP Blocks

Table A.3: List of IPs used in the system implementation

<table>
<thead>
<tr>
<th>IP</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI GPIO</td>
<td>2.0.9</td>
<td>General Purpose Input/Output core provides a general purpose input/output interface to the AXI interface.</td>
</tr>
<tr>
<td>AXI Uartlite</td>
<td>2.0.11</td>
<td>Generic UART (Universal Asynchronous Receiver/Transmitter) with an AXI Interface</td>
</tr>
<tr>
<td>Clocking Wizard</td>
<td>5.2.1</td>
<td>The Clocking Wizard creates an HDL file that contains a clocking circuit customized to the user’s clocking requirements.</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>9.5.3</td>
<td>The MicroBlaze 32 bit soft processor core, providing an instruction set optimized for embedded applications with many user-configurable options such as Instruction and Data-side cache with AXI interfaces, Floating-Point unit, Memory Management Unit, and fault tolerance support.</td>
</tr>
<tr>
<td>LMB BRAM Controller</td>
<td>4.0.7</td>
<td>Local Memory Bus (LMB) Block RAM (BRAM) Interface Controller connects to an Lmb bus</td>
</tr>
<tr>
<td>Local Memory Bus (LMB) 1.0</td>
<td>3.0.1</td>
<td>The LMB is a fast, local bus for connecting MicroBlaze I and D ports to peripherals and BRAM</td>
</tr>
<tr>
<td>Constant</td>
<td>1.1.2</td>
<td>Gives a constant signed value</td>
</tr>
<tr>
<td>Block Memory Generator</td>
<td>8.3.1</td>
<td>The Xilinx LogiCORE IP Block Memory Generator replaces the Dual Port Block Memory and Single Port Block Memory LogiCOREs. It supports RAM and ROM functions over a wide range of widths and depths.</td>
</tr>
<tr>
<td>MicroBlaze Debug Module</td>
<td>3.2.4</td>
<td>Debug module for MicroBlaze Soft Processor.</td>
</tr>
<tr>
<td>Processor System Reset</td>
<td>5.0.8</td>
<td>Processor Reset System</td>
</tr>
<tr>
<td>AXI Crossbar</td>
<td>2.1.8</td>
<td>The AXI Crossbar IP provides the infrastructure to connect multiple AXI4/AXI3/AXI4-Lite masters and slaves.</td>
</tr>
</tbody>
</table>
A.3 Design Automation

Hardware and Software development have been generated using Network-on-Chip System Generator (NGS) [72]. The generated system produces necessary files and folders for the IPs in a structure format compatible with Xilinx ISE design suite. The IPs have been restructured according to [77] to be compatible with Vivado Design Suite [78, 79] and guide in [80].

The specific versions of tools that have been used are shown in Table A.4.

Table A.4: Versions of tools used in the design and development

<table>
<thead>
<tr>
<th>Tool name</th>
<th>By</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vivado Design Suite</td>
<td>Xilinx</td>
<td>2016.2</td>
</tr>
<tr>
<td>Xilinx Software Development Kit</td>
<td>Xilinx</td>
<td>2016.2</td>
</tr>
<tr>
<td>NoC System Generator</td>
<td>KTH</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Appendix B

NoC Synthesis and Characterisation

Generally, packet propagation latency has an inter-dependent relation with the activities of the network and packet length in units of words (a flit carries one word of info). But due to the time-triggered operation of the NoC, the dependence on network activities is negligible and therefore, the packet latency, $T_{packet}$, follows this equation.

$$T_{packet} = T_{Setup} + T_{flit} \times \text{flits} \quad (B.1)$$

Or,

$$T_{flit} = 2T_{NI \text{ to NoC}} + T_{Switch \text{ to Switch}} \times N_{switches} \quad (B.2)$$

Where:

- $T_{flit}$: Total propagation latency for one flit from sending NI to receiving NI.
- $T_{Setup}$: Setup time required before the actual flits can be sent.
- $T_{NI \text{ to NoC}}$: Flit propagation latency within the NI until the flit is injected at the Switch.
- $T_{Switch \text{ to Switch}}$: Flit propagation latency within the switch receiver and transmitter until the flit is injected at the subsequent Switch or NI.
- $N_{switches}$: The number of switches the flit has to traverse to reach the destination NI. Also known as the number of hops or hops counter.
- $\text{flits}$: Packet size in term of number of flits or words, $W$.

The NoC timing parameters for NI to Switch, Switch to Switch and Switch to NI are reported in Table B.1. $T_{NI \text{ to Switch}}$ encapsulates the worst case delay imposed by the time-trigger mechanism of the NoC in addition to the access time to fetch data from the memory. $T_{Switch \text{ to Switch}}$ includes the timing in the switch’s receiver and transmitter. $T_{Switch \text{ to NI}}$ includes time to fetch data from the switch and store it at the right memory address. Those
Table B.1: NoC timing parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{NI \text{ to } Switch}$</td>
<td>16</td>
</tr>
<tr>
<td>$T_{Switch \text{ to } Switch}$</td>
<td>4</td>
</tr>
<tr>
<td>$T_{Switch \text{ to } NI}$</td>
<td>4</td>
</tr>
</tbody>
</table>

numbers can be used to formulate timing analysis for various NoC topologies. For 2x2 NoC, the maximum delay can be taken from diagonal nodes can thus be described as follows:

$$T_{\text{flit}} = 32 \times W \quad \text{(B.3)}$$

$$T_{\text{packet}} = 64 + 32 \times W \quad \text{(B.4)}$$

Where $T_{\text{packet}}$ is in the units of cycles and $W$ is the packet length in words (1 word is 32 bit). The 64 cycles are actually for the first two flits which are used to relay the time at which the packet was inserted and the length of the packet. For packet size of one 32-bit word, $T_{\text{packet}} = 96$ cycles.

Implementation wise, the design was synthesized at 50MHz on "xc7z020 clg484-1" chips. Reports on resource utilisation, timing, energy consumption are extracted from Vivado. The longest path delay is found on the routing table that is used for extracting the destination process ID based on the channel source. This signifies a bottleneck for larger designs.

The packet propagation latency in the NoC, energy consumption per flit, and resource utilization are shown in Table B.2 and Table B.3. The tables also give statistics for CPU (Xilinx Microblaze) and its embedded memory (64KB).

Table B.2: Resource utilisation

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Memory</th>
<th>NoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>1115</td>
<td>7</td>
<td>2768</td>
</tr>
<tr>
<td>Registers</td>
<td>2272</td>
<td>13</td>
<td>6723</td>
</tr>
<tr>
<td>BRAM</td>
<td>0</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

The static power consumption is derived by multiplying the resource utilisation ratio of the NoC times the whole device resources. The dynamic power consumption per module is provided directly from Vivado simulation. At this frequency the estimated dynamic power consumption is 0.012
Table B.3: Performance metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>NoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Power (Watt)</td>
<td>0.012</td>
</tr>
<tr>
<td>Static Power (Watt)</td>
<td>0.006966</td>
</tr>
<tr>
<td>Latency Per Transaction (Cycles)</td>
<td>32</td>
</tr>
<tr>
<td>Latency Per Transaction (micro Sec.)</td>
<td>2.777</td>
</tr>
<tr>
<td>Energy Per Transaction (mJ)</td>
<td>52.684</td>
</tr>
</tbody>
</table>

Watt/Flit. The dynamic power consumption of the NoC depends on the number of switches which is 0.003 Watt/switch. Note that this power estimation results are of low confidence and are used for qualitative and comparative analysis.