TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Integrierte Systeme

Network on Chip Interface for Scalable Distributed Shared Memory Architectures

Muhammad Aurang Zaib

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzende/-r:		Prof. Dr.-Ing. Ulf Schlichtmann
Prüfende/-r der Dissertation:	1.	Prof. Dr. sc.techn. Andreas Herkersdorf
	2.	Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
	3.	————

Die Dissertation wurde am 21.06.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 18.01.2018 angenommen.

Abstract

Five decades ago, Gordon Moore predicted the doubling of transistors per unit chip area every 12 months. Until the present day, the semiconductor industry has been successful in following Moore's prediction which has resulted in the form of state of the art System on Chip (SoC) architectures. This advancement in the semiconductor design technology has strongly influenced the architecture of individual components in a System on Chip. The biggest example is the transformation in the Central Processing Unit (CPU) architecture. Till the last decade, the higher clock frequency and the micro-architectural enhancements were considered to be the possible means for getting better performance from central processing units. Micro-processors up to multi-gigahertz clock frequency and out-of-order execution support are the prime evidence of the previous design trends. However, physical and architectural limitations for producing more sophisticated single core processing systems have resulted in a paradigm shift, which advocates the deployment of multiple cores in the modern state of the art SoCs. This paradigm shift has also revolutionized the way in which communication infrastructure and memory hierarchy are realized in modern many-core architectures. Distributed interconnects like Network on Chip (NoC) have replaced the conventional bus-based communication to match the scalability requirements. In addition, memory is physically distributed in the architecture to circumvent data access bottlenecks. However, better design productivity and bounded Time-to-Market advocate the support for legacy shared memory applications in many-core architectures. Therefore, Distributed Shared Memory (DSM) architectures are introduced, which support both shared and distributed memory programming models and hence deliver a good compromise between performance and productivity.

The emergence of Network on Chip based DSM architectures brings forward major challenges for efficiently exploiting the available parallelism in modern computing systems. The first major challenge is the efficient management and utilization of Network on Chip communication infrastructure. The management of Network on Chip refers to the allocation of communication resources between connection-oriented and connectionless traffic. In DSM architectures, the communication patterns between different processing nodes are highly dependent on application mapping and memory hierarchy. In addition, these patterns may vary at run-time depending on the application's communication behavior. Conventional communication resource management methodologies result in sub-optimal performance and high power consumption because they do not regard the above-mentioned factors which affect run-time network traffic. Therefore, strategies are required which offer optimized utilization of communication resources in distributed shared memory architectures during execution of applications. The second important challenge is the requirement of appropriate synchronization mechanisms when the application is mapped on architectural nodes, which are interconnected via Network on Chip. State of the art methods are heavily dependent on the system software to manage synchronization between remote nodes. In addition, these methods disregard communication latencies over

Network on Chip during the interaction of remote software instances. Therefore, in the scenarios where the software layers expand the computation on remote processing nodes, the synchronization delays lead to high performance overhead and hence mitigate the advantages of task level parallelism.

In order to address the two above-mentioned challenges, the design of the Network Interface (NI) gains importance. The network interface plays the role of a gateway between processing cores (computation) and Network on Chip (communication). In this work, a network interface architecture is presented, which proposes a novel self-optimization based mechanism for run-time communication resource management. The proposed self-optimization strategy reduces the communication latencies for applications and decreases the energy consumption by improving utilization of Network on Chip resources in DSM architectures. In order to address the synchronization overhead, the network interface architecture is extended to offer hardware support for software instances which communicate over the Network on Chip. As the case studies for the synchronization support, data transfer and task spawning operations between tiles are considered. A novel concept is presented, which manages complete handshaking required during remote data transfer operation in hardware. In addition, an innovative approach for task spawning is proposed which offloads the software from handling synchronization events during task spawning.

Simulation and FPGA prototyping frameworks are developed to validate the claims of proposed methodologies. Real world case studies including video processing and matrix multiplication applications are used to compare the performance of presented network interface architecture with the state of the art designs. The evaluations related to communication resource management show a reduction in average network packet latency by 35% with the proposed concept when compared with the state of the art approach. In addition, our methodology reduces the power consumption of the communication infrastructure by up to 33% for MPEG video processing application. The presented hardware support for synchronization brings an improvement of 38% in comparison to the reference approach for the investigations related to remote task spawning. The hardware area cost of the proposed enhancements is around 16% of the basic network interface size for FPGA and ASIC platforms.

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Dr.sc.techn. Andreas Herkersdorf. He considered me to be the part of the elite group of researchers by giving me the chance to work on this great research topic. In addition, he provided me his valuable advice, inspiration, and encouragement throughout my Ph.D. study. I am truly grateful for his help, not only in my research but also in my life. I have the highest regard for the support that he provided me as a Mentor during my stay at the Institute. In addition, I would also like to thank Prof. Dr.-Ing. Walter Stechele for his guidance and support during my Ph.D. Additionally, I am grateful for his supervision during my work as a student in the scope my internship and Werk student activities.

I would like to thank Prof. Jürgen Becker for being the co-examiner of my thesis and for his valuable comments. I also want to thank Prof. ... for chairing the examination committee.

In particular, I would like to thank Dr.-Ing. Thomas Wild, for the constructive discussions and valuable inputs throughout my Ph.D. I truly believe that without his constant support, the successful completion of my work would have been far more difficult. I am grateful to the Invasive Computing colleagues for their support during my work in the course of the Invasive Computing project. During my work in the Invasive Computing project, I was fortunate to work with a group of talented and creative colleagues from different institutes. At the first place, I would like to name Jan Heisswolf and Andreas Weichslgartner. It was a tremendous experience for me to work together with them in the B5 sub-project. In addition, working together as a team with Benjamin Oechslein, Jens Schedel, Gabor Dresher, Christoph Erhardt, Manuel Mohr and Stephanie Friedrich for integrating and prototyping was a great experience. All the above-mentioned colleagues expertise in different aspects of embedded systems design and the discussions with them improved my professional skills in those aspects.

A lot of thanks goes to my colleagues and friends at LIS. LIS colleagues include Dirk Gabriel, Thomas Goldbrunner, Christian Herber, Erol Koser, David May, Andreas Oeldemann, Felix Miller, Andre Richter, Preethi Parayil Mana Damodaran, Johny Paul, Ravi Kumar Pujari, Michael Vonbon and Philipp Wagner and Stefan Wallentowitz. In particular, I want to thank Ravi Kumar Pujari, Johny Paul and David May for the constructive discussions in Invasive Computing project meetings. I would also like to thank Stefan Wallentowitz and Preethi Parayil Mana Damodara for their meaningful contribution towards my work in Memory Hierarchy project meetings. Their comments helped in improving my work. I also want to thank the members of the LIS Institute administration for maintaining an excellent workplace and providing me their constant support. LIS administration staff includes Verena Draga, Wolfgang Kohtz, Gabi Spörle, and Doris Zeller.

Last but certainly not the least, I would like to thank my parents, my wife and my daughter, to whom this thesis is dedicated. My parents raised me up and tried their best to provide me the best possible education and a healthy atmosphere at home. Especially, my mother who provided me constant emotional support for all my ambitions in life. I want to thank my lovely wife Huma for her love, constant support, encouragement, and understanding during my work. My daughter has been the biggest source of the motivation for me to progress in life since last 4 years.

Munich, June 2017
Muhammad Aurang Zaib

To my Parents, wife and daughter

Contents

1. Introduction

Since five decades, the advancement in the semiconductor design technology has increased the number of transistors on a single chip. Gorden Moore already predicted this trend in the form of famous Moore's law in 1965 [123]. According to this law, the number of transistors on an integrated circuit doubles every 18 to 24 months. Figure 1.1 shows the transistor count in state of the art processors with increasing years. The figure shows clearly that the semiconductor industry has been successful in preserving Moore's prediction till the present year.

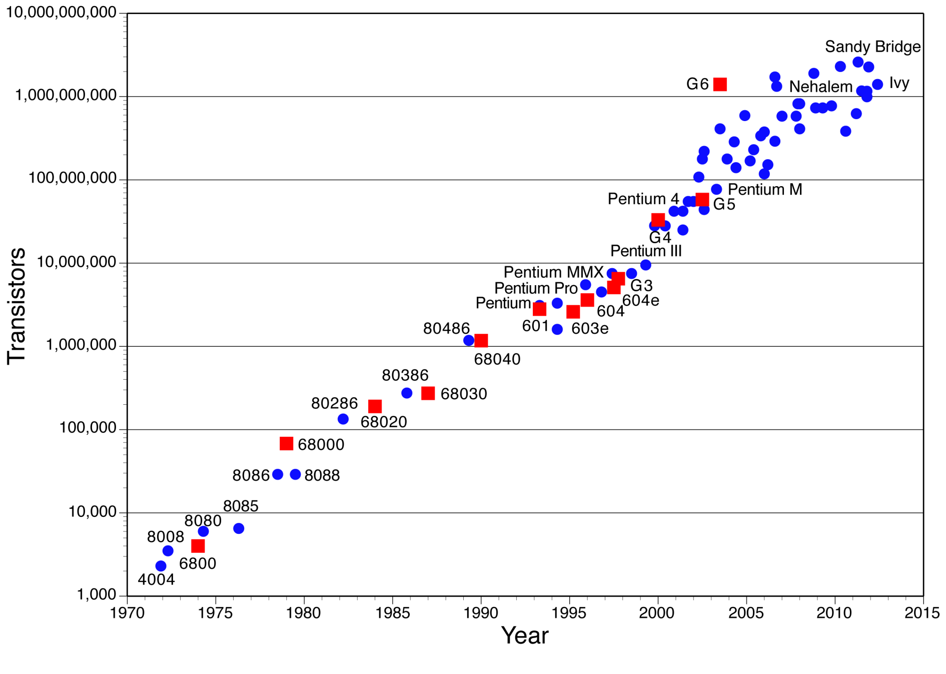


Figure 1.1.: Increasing transistor count on single chip according to Moore's law [129]

Due to the successful evolution of the chip design technology, semiconductor manufacturers have been able to support continuously increasing scale of integration. Intel Core i7 processors, which are based on 5th generation Broadwell-U micro-architecture, are fabricated using 14 nm technology [138]. This processor was introduced in January 2015 and consists of 1,3 Billion transistors representing the state of the art advancement in digital design and fabrication. In general, it can be stated that the continuous evolution of semiconductor industry is exhibited through the release of state of the art integrated circuits with every coming year.

The advancement in semiconductor technology comes up with new challenges for chip designers. Technology growth is responsible for some of these challenges, which are manufacturing complexity, process variability, and static power dissipation. Another important aspect, which deals with the system level design, is the increasing gap between the technology growth and the design capabilities [61]. State of the art design tools, processes and methodologies are considered to be the reason because of which the design productivity is unable to match pace with the technology advancement. As a result, system developers are continuously working on novel approaches to address these concerns in order to prevail the trend of producing powerful integrated circuits.

1.1. Motivation

In general, the introduction of every state of the art integrated circuit offers an opportunity for the application developers to generate better products. As a consequence of the continuous evolution of modern integrated circuits, the electronic devices around us are revolutionized. In the past, the functionality which was realizable on printed circuit boards can now be fabricated on a single chip. Therefore, bigger machines are replaced by smaller sized modules. This trend towards miniaturization of electronic devices with the passing of the years is illustrated in the figure 1.2.

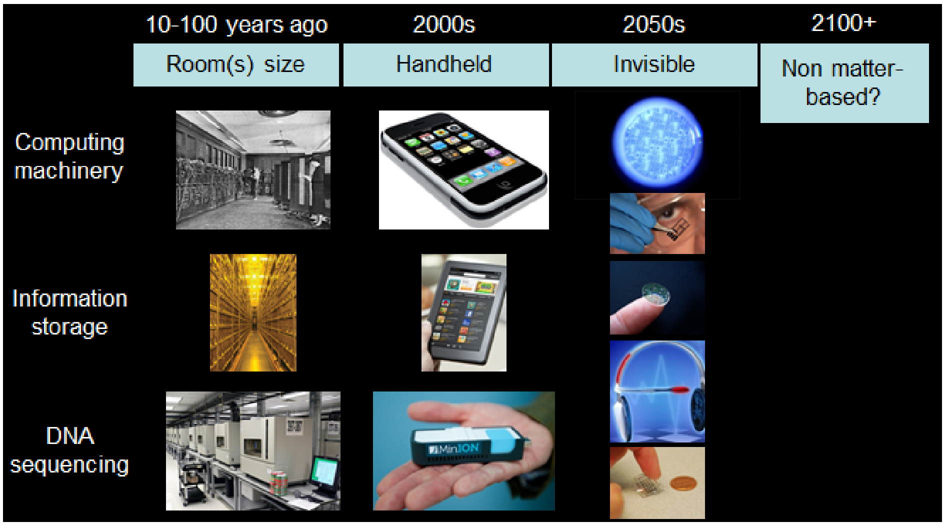


Figure 1.2.: Miniaturization of electronic devices based on integrated circuits [128]

The figure shows that handheld smartphones can perform the functionality which was carried out by the big computing machines a few decades ago. Nowadays, the laptops possess computing power which used to be possible on computing server in the past.

Wrist watches and gadgets are equipped with the features, which were difficult to realize on computers previously. This miniaturization trend in electronic devices has triggered the innovation in commercial industry as well. This is due to the fact that in many industries, a major share of the technology comes from electronics. For example, the main advancement in automobile industry comes from electronics in form of navigation, distance control or camera-based driver assistance [40] [71].

It is desired that this trend carries on in the future and hence leads to more and more functionality per unit chip area. The wish to achieve this trend is also depicted in the figure 1.2, where we see the current state of the art devices being replaced by surprisingly small modules in future. This keeps the semiconductor designers and manufacturers motivated to come up with novel ideas and methodologies for designing next-generation integrated circuits.

1.1.1. System on Chip

As stated in the previous section, increasing integration densities have resulted in reducing the sizes of electronic devices. This has mainly occurred because the basic building blocks of digital design like processors and memories, could be packed in a smaller chip area [5]. A System on Chip can be defined as an integrated circuit that consists of one or more processing cores connected via an on-chip interconnect. In addition, it contains on-chip memory for storing frequently accessed data and I/O interface(s) for communicating with the external world. More advanced SoCs may also consist of components like hardware accelerators which are incorporated to improve the execution of specific tasks. Figure 1.3 illustrates an example of a System on Chip with different interconnected components.

Researchers have proposed an approach, which emphasizes the reuse of the existing Intellectual Property (IP) blocks for System on Chip design. This methodology is referred as platform-based design [122]. Platform-based design approach makes System on Chip more suitable as compared to custom ASICs because they can deliver higher performance with medium development effort. Therefore, SoCs are a preferred choice for investigating novel computing concepts when realized on suitable prototyping platforms. Traditionally, simulation and FPGA prototyping are used by developers to debug and verify the functionality of System on Chip.

1.1.2. Multi and Many-core Architectures

The increasing number of transistors has been utilized in the past to improve the performance by increasing the complexity of SoCs. The introduction of complex instruction set architecture, branch prediction and out-of-order execution are some of the examples where the IC complexity is increased to get better performance. Increased transistor switching speed resulting from the technology miniaturization offered the second major way to get better execution time. As a result, the clock frequency at which the integrated circuits can be operated on, steadily increased. Therefore, micro-architectural enhance-

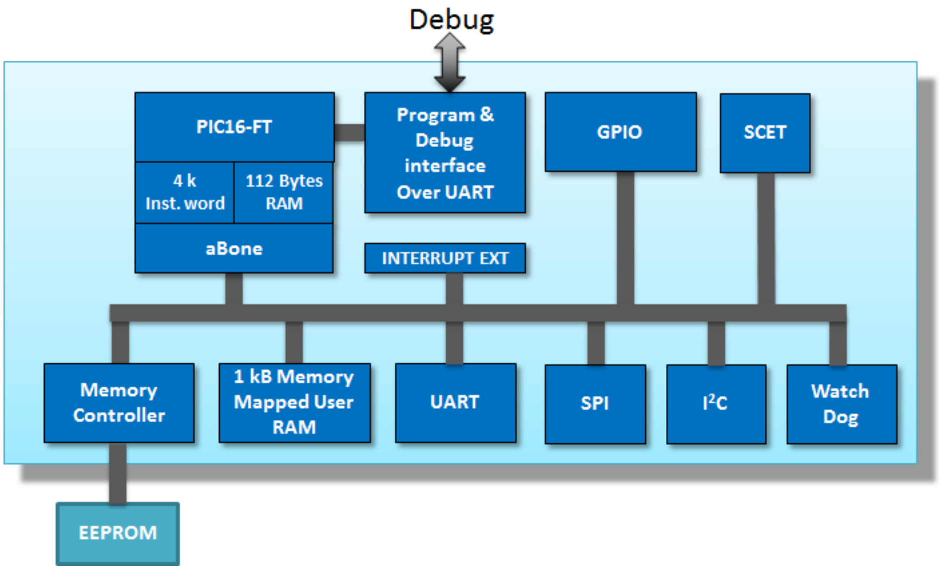


Figure 1.3.: System on Chip based on OpenPIC softcore processor [92]

ments and increased clock frequency played an important role in the development of state of the art processing systems.

In recent years, it was observed that the micro-architectural extensions and technology improvement could hardly be used for further increasing the performance of single cores. According to Pollack's rule, the increase in design complexity of underlying hardware architecture does not guarantee a linear increase in system performance [18]. Besides limited performance improvement, increasing the clock frequency results in generating more complex and power hungry designs. This is due to the fact that the increasing clock speed results in the transistors to switch faster and hence causing more power consumption. In addition, the static power dissipation becomes the significant contributor to the overall power dissipation with the shrinking transistor sizes [127].

Another important concern is the increase in wire latencies with respect to the gate delays for advanced manufacturing technologies, which eventually put a limit on the achievable clock frequency [84]. Keeping in view these aspects, it can be stated that the state of the art transistor technologies have reached the limit where it is nearly impossible to realize more powerful single core processors [46]. In addition, architectural enhancements like superscalar pipelining for exploiting Instruction Level Parallelism (ILP) are not suitable for a wide range of applications. Many real world applications benefit from Task Level Parallelism (TLP), where tasks/threads are executed on multiple independent execution units to increase the system's overall performance.

All above-mentioned facts have led to the trend of building computing systems based on multiple processing cores. The multi-core systems must not necessarily be clocked at the same frequency as the state of the art single core processors, but they improve overall performance by exploiting task level parallelism. In addition, multi-core architectures result in reduced design complexity and power consumption. Intelligent methodologies are being researched for many-core platforms to save the power by turning off idle cores at run-time [35]. Approaches like Dynamic Voltage Frequency Scaling (DVFS) and load balancing can be applied to keep uniform heat dissipation across the chip. The uniform heat dissipation improves the system reliability and reduces leakage currents.

Intel's Xeon Phi Coprocessor [74] and Tilera's Tile-Gx100 [120] are state of the art examples of existing many-core architectures. Tiled many-core architectures are introduced by incorporating the concept of platform-based design to improve the design productivity. In addition to the above-mentioned commercial outcomes, significant work has been done by the researchers in academia to develop novel frameworks for enabling rapid exploration of future many-core architectures [12], [135].

1.1.3. Distributed Shared Memory Architectures

The trend towards introducing multi and many-core processors has also affected the communication requirements of System on Chip. Conventional bus based systems have centralized arbitration mechanisms which lead to the starvation problems for the connected nodes. Starvation implies the scenario where free nodes are denied the access to the bus because of an already in-progress transaction. In addition, bus-based architectures carry a significant cost in form of the number of wires, which are required to connect a relatively large number of components. Keeping in view the above-mentioned concerns, shared bus based communication infrastructures are not considered as an appropriate choice for many-core architectures. Distributed interconnects like Network on Chip are found to be more suitable for many-core systems [7]. Distributed interconnect affects the communication between tasks, which are mapped on different tiles. Therefore, the application architects have to think in advance about the communication behavior and the memory access patterns of their program. As a result, the choice of the programming model in terms of memory abstraction, shared memory or message passing, becomes an important aspect of communication infrastructure design.

In shared memory model, the communication happens transparently from the programmer's perspective. Therefore, the shared memory application developer is not much bothered about the communication related aspects like data access patterns and the delays involved in fetching data. Before the emergence of many-core systems, much of the legacy code was written by keeping in view shared memory model. On the other side, message passing is established in distributed memory systems. In such systems, the communication is not completely transparent for the programmer. Depending on the locality of data, different communication mechanisms have to be adopted by the application programmer in message passing systems. Keeping in view the different characteristics of both programming models, it is clear that each of them poses different challenges towards the design of underlying many-core architecture. Till the last decade, application developers used to deploy shared memory model predominantly for executing their applications over single chip platforms. The motivation behind this was the possibility to reuse legacy application code, developed for conventional x86-based shared memory architectures and as a result improving design productivity. On the other side, the message passing based communication approach was only deployed in high-performance computing clusters in the past. However, with ever increasing computational parallelism and novel communication methodologies, message passing has emerged as a scalable programming model for single chip architectures in recent times.

Another important aspect of communication infrastructure design is the wide range of variation in communication behavior of modern real world applications. The communication behavior of such applications is strongly influenced by the characteristics of the underlying hardware platform. Therefore, it is difficult to strictly classify applications whether they benefit from either shared memory or message passing based programming model before they are actually executed on the given architecture. On the other side, modern many-core systems are expected to support the execution of a wide range of applications, each of which may benefit from either a shared memory or message passing based programming style. These aspects regarding co-design of application and architecture motivate to look for a flexible methodology for supporting shared memory and message passing programming models on many-core architectures.

In order to support both shared and message passing-based execution, researchers have introduced distributed shared memory architectures [103]. Both shared memory and message passing-based programming models can be efficiently supported on a DSM architecture. DSM architectures enable the shared memory programming paradigm by allowing access to all memories present in the system through a global address space. In addition, by distributing the memories in different nodes of architecture, the message passing-based programming model is supported. As compared to a conventional distributed memory system, a DSM architecture provisions access to each memory in the system through Non-Uniform Memory Access (NUMA) manner. Partitioned Global Address Space (PGAS) programming model has come up as a preferable choice for programming distributed shared memory architectures [142]. Both DSM and PGAS terminologies are used in close conjunction with each other. However, for the sake of clarity, it is important to mention here that the DSM refers to the type of architecture where the memories are physically separated [139]. Whereas, the PGAS is a programming model which assumes that the entire memory of the system is accessible through a global address space. However, the address space itself is logically partitioned [141].

An example configuration of a single chip distributed shared memory architecture is shown in the figure 1.4. In a many-core DSM architecture, Network on Chip is deployed as a distributed interconnect which connects different nodes. The network interface is a component which joins the System on Chip blocks within a tile to the NoC router. In a DSM architecture, the physically shared memory may also be present in the form of off-chip memory and could be realized as one or more memory tiles. In addition, the on-chip memory is physically distributed among different nodes in the architecture as tile local memory. Besides supporting shared memory and message passing programming models, the purpose of distributing memory in different architectural nodes is to prevent the creation of data access hotspots.

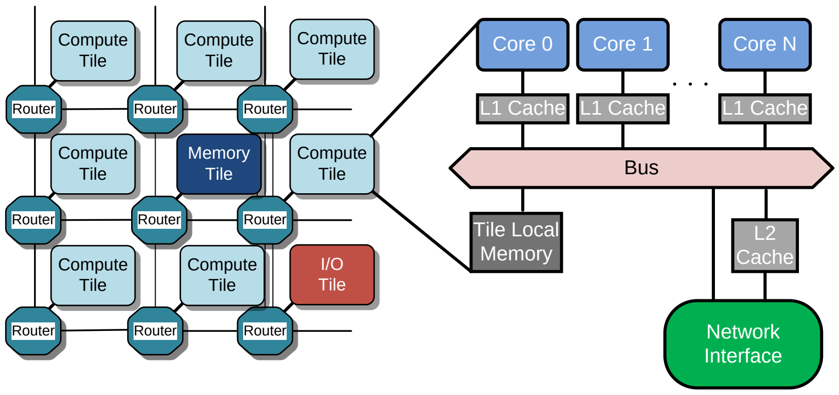


Figure 1.4.: Network on Chip based DSM architecture

1.2. Challenges in DSM Architectures and our Contributions

Distributed shared memory architectures are realized following a system-level design approach to support high computational parallelism in modern many-core systems. However, the performance gain which can be achieved on such computing platforms depends heavily on the synergy between the application and the architecture. In particular, performance improvement on the distributed architecture is limited by the part of the application which runs in parallel on multiple cores, following Amdahl's law [65]. In the same direction, the advent of DSM architectures not only comes up with better performance and increased designer productivity, it also brings forward additional challenges for system designers. Distributed communication infrastructure, physically partitioned memory hierarchy and the corresponding application mapping at run-time expose two major challenges for the developers of DSM architectures:

- Distributed shared memory architectures define a memory hierarchy which is different as compared to the conventional state of the art architectures. The specific memory hierarchy with physically distributed memory emphasizes the need for appropriate application mapping strategies on the underlying many-core architecture. However, the mapping of application tasks on the underlying processing resources may vary depending on the application mapping algorithm and architecture status at run-time. The variation in task allocation on processing resources results in continuously changing bandwidth requirements. As a result, communication patterns are generated which can not be predicted statically. Network on Chip is deployed as communication infrastructure in distributed shared memory platforms. Efficient management of Network on Chip requires appropriate distribution of its resources between connection-oriented and connectionless traffic. State of the art communication resource management mechanisms can not react on run-time traffic changes. Therefore, these approaches result in degraded application performance and inefficient utilization of communication infrastructure.

- Distributed nature of DSM architectures leads to a high amount of data communication between different software instances which are mapped on remote processing nodes. The remote processing nodes are interconnected via Network on Chip. The communication between remote nodes includes data transfer as well as the handshaking between distributed operating system instances to manage/coordinate the execution of applications on underlying processing resources. However, the communication delays between different application instances, which are running on different tiles, are large because of the transmission latencies over NoC. State of the art communication and synchronization mechanisms disregard the characteristics of distributed communication infrastructure and hence lead to high performance overhead.

For real world applications to benefit from the available parallelism in distributed shared memory architectures, it is essential to address above-mentioned challenges [64]. Therefore, in the scope of this work, we have made contributions for tackling these problems.

- In order to address the first issue, we have proposed a novel concept which gathers the communication history between remote tiles in the network interface and then utilizes it to establish end-to-end connections. This concept enables optimized management of communication resources by keeping in view the dynamic communication patterns. Our investigations show that the proposed approach results in reduced communication latency as well as lesser energy consumption when compared to state of the art methodologies [20].
- As the second contribution of this work, efficient mechanisms are introduced for reducing the communication and synchronization overhead for the software instances running on remote tiles. As a case study, data transfer and task spawning operations between remote tiles are considered. The software is offloaded from synchronization duties by introducing hardware support. The introduced hardware extensions result in reducing the overall execution time of applications as compared to state of the art reference [78].

Both contributions of our work are realized as hardware extensions in the network interface design. Therefore, the outcome of this work is a network interface architecture which is equipped with novel concepts for efficient communication resource management and hardware support for communication and synchronization between remote tiles. The proposed research contributions have been made in the scope of Invasive Computing [131]. Invasive computing incorporates the concept of resource-awareness for the management of the future many-core system. Further details about this novel research paradigm would be provided in chapter 2. It is worth mentioning that our contributions are not confined to invasive computing and can be applied to state of the art many-core systems as well.

1.3. Outline

The work is organized as follows. The background and related work of this work are explained in chapter 2. State of the art concepts related to on-chip communication are presented in sections 2.1 and 2.2. Bus-based systems as a predecessor of Network on Chip are discussed in section 2.1. Network on Chip is introduced in section 2.2. Basic Network on Chip components including the network interface and state of the art NoC architectures are also discussed in the same section. Conventional communication support in the network interface including mechanisms like protocol translation, shared memory access, remote direct memory access and Quality of Service support, is presented in section 2.3. The context of this work is discussed in section 2.4. In particular, the concept of invasive computing and the major software layers are described in section 2.4.1 and 2.4.2 respectively. In section 2.4.3, the individual components of the heterogeneous InvasIC architecture are presented.

Chapter 3 describes the proposed concept. Section 3.1 presents the approach to optimize the communication resource utilization in DSM architectures. Section 3.2 narrates the proposed support for communication and synchronization between remote software layers. The implementation details related to the proposed network interface architecture are addressed in chapter 4. The layered architecture model of the network interface representing the modular design approach is described in section 4.1. Section 4.1 also presents the state of the art communication support in the network interface architecture. The implementation related to the proposed concepts is described in section 4.2 and 4.3.

The details about the experimental setup and evaluations are provided in chapter 5. Sections 5.1 and 5.2 provide the details about the RTL simulation and FPGA prototyping frameworks. The validations of the proposed concepts on the respective simulation and prototyping frameworks are provided in sections 5.3 and 5.4 respectively. Chapter 6 gives the conclusion.

2. Background and Related Work

The modern Multiprocessor System on Chip (MPSoC) architectures consist of multiple processing cores, customized hardware accelerators, on-chip and off-chip storage and I/O interface(s). The components in an MPSoC architecture need to interact with each other during the execution of applications. The communication may occur as a result of different operations, e.g. data exchange in the form of either reading instructions from memory or writing data to memory or synchronization messages for signaling various events. Because of the presence of multiple processing instances, the overall performance of the parallel architecture heavily relies on the throughput supported by the communication infrastructure.

MPSoC communication architectures exist in various configuration and topologies. However, in the scope of this work, we have considered two types of communication architectures; 1) Bus-based communication architectures, 2) Network on Chip. It is done because our proposed network interface architecture acts as a bridge between the two above-mentioned communication architectures and the understanding of these communication infrastructures is important for our concept. Bus-based designs lead towards the trend of IP reuse and hence are very suitable to realize modern System on Chip as tiled many-core architectures. On the other hand, Network on Chip addresses the required scalability for many-core systems. The details about the two relevant SoC data exchange methodologies is provided in the following sections.

2.1. Bus-based Communication Architectures

Buses are one of the most widely used means of communication on System on Chip architectures because of their simplicity and effectiveness. There could be multiple physical realizations of a bus, e.g. single shared bus, hierarchical bus etc. Each realization offers its own advantages and disadvantages. In single shared bus topology, several components are connected via a shared channel, hence reducing the implementation cost of interconnect. The hierarchical bus consists of several shared buses which are interconnected by bridges to create a hierarchy. A bus protocol/standard defines the characteristics of a communication transaction. The goal of the bus protocol is to provide a bus architecture specification, which is independent of the technology to enable IP reuse. Bus Master is the component which starts a transaction whereas the slave corresponds to the component which responds to the transaction initiated by the bus master. In addition, the bus protocol defines the arbitration, which is necessary to determine the priority of access if multiple requests arrive to use the shared medium.

In the past, several on-chip bus architecture standards have been proposed to address the communication requirements of modern SoCs [111]. Some of the popular standards are

ARM Microcontroller Bus Architecure (AMBA) [39], IBM Coreconnect [70], STMicroelectronics STBus [91], Opencore Wishbone [109], and Altera Avalon [25]. The master/slave components which adhere to the same bus standard can be seamlessly integrated into the architecture. This speeds up the SoC development by following the platform-based design approach through reusing the IP blocks compliant to the deployed bus protocol. AMBA is the most widely used on-chip communication standard. It consists of different sub-protocols, which correspond to bus architectures requiring different communication bandwidth. Widely known sub-protocols are Advanced Peripheral Bus (APB), Advanced High-Performance Bus (AHB) and Advanced eXtensible Interface (AXI). In this work, the details specific to AHB Protocol are described because this is the standard which we have used in our work.

2.1.1. Advanced High-Performance Bus (AHB)

AHB bus standard is designed to interface components which require high communication bandwidth. Components like microprocessors, DMA and memory controllers are the suitable candidates to be interconnected via AHB bus. A typical AMBA AHB bus system is shown in the figure 2.1. In the figure, ARM processor and DMA accelerator are connected to the AHB bus as master modules. Slave components are present as an on-chip RAM and off-chip memory interface. Relatively low bandwidth components like UART, Keyboard etc, are connected over an APB bus. Both Buses are connected via an AHB-APB bridge, which also acts as a slave component.

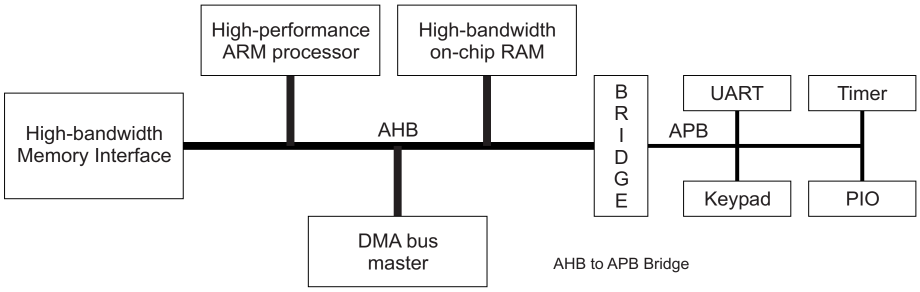

High-performance ARM processor
High-bandwidth on-chip RAM
High-bandwidth Memory Interface
AHB
BRIDGE
APB
UART
Timer
Keypad
PIO
DMA bus master
AHB to APB Bridge

Figure 2.1.: A typical AMBA AHB-based System [126]

As stated above, an AHB bus system may consist of multiple components. In the following text, roles of different bus components are described keeping in view shared bus topology:

- **AHB Master :** An AHB bus master initiates read and write transactions by providing the associated address and control information. In the case of shared bus, only a single bus master is allowed to actively use the bus at a given time.

- **AHB Slave :** Bus slave answers the read or write transaction when the corresponding transaction lies in its address range. In addition, the bus slave signals the success, failure or waiting status of the transaction through control signals.
- **AHB Arbiter :** An AHB bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers over the shared bus. Arbitration policies considering priority or access fairness can be defined according to application requirements.
- **AHB Decoder :** The AHB decoder is used to decode the address contained in each transfer and provide a select signal for the corresponding slave, which is the destination of the transfer.

Bus masters use a single address bus to transmit the addresses of the slaves to complete the corresponding read and write transactions. The size of the address bus defines the address space. Till the recent past, 32 bit wide address was considered sufficient for SoC processing requirements. However, embedded processor with 64 bit address space have been introduced in the last few years to meet the processing requirements in high-end applications [140] For data bus, AHB offers separate buses for read and write transactions. It is done to avoid the use of buses as tri-state implementation keeping in view AMBA specifications [126]. The width of data buses is configurable from 8 bits to 1024 bits, depending on the SoC bandwidth requirements. The brief description of the important AHB signals is provided in Appendix A.1.

The AHB bus offers following distinct features to support high bandwidth and low latency data transfers:

- **Burst Transfer :** A burst operation is defined as a data transaction, initiated by a bus master, to read or write multiple words of same data size. Burst transactions to the consecutive addresses are referred as incremental bursts. When the start location of the transaction is not aligned to the total size of the burst, such transfers are called wrapping bursts. The number of words which are transferred in a single burst transaction is referred as a beat. Four, eight, sixteen beat as well as bursts of infinite length are supported on AHB bus. Burst transfers increase the data transfer efficiency by transferring a large amount of data in a single transaction. Therefore, the burst transfers are particularly useful in the applications like direct memory access. Figure 2.2 shows the example of an incremental four-beat burst transfer on AHB bus.
- **Split Transfer :** Split transfers improve the overall utilization of the bus by separating each read or write operation in two phases. The operation of the master providing the address to a slave is named as Request phase whereas the slave responding with the corresponding answer is called Response phase. Split transfer feature separates the execution of request and response phases. Therefore, in the time when the master which initiated the request waits for its response, other masters can execute their requests on the bus. Hence, the blocking of the bus for the entire duration of read or write operation can be avoided. Split transfers lead to better bus utilization especially in the scenarios where slave component requires more time to return the response. In addition, the split transfer support helps to resolve different deadlock scenarios on AHB bus. These deadlocks may occur if the master which initiated the request, holds the bus for the infinite duration of time.

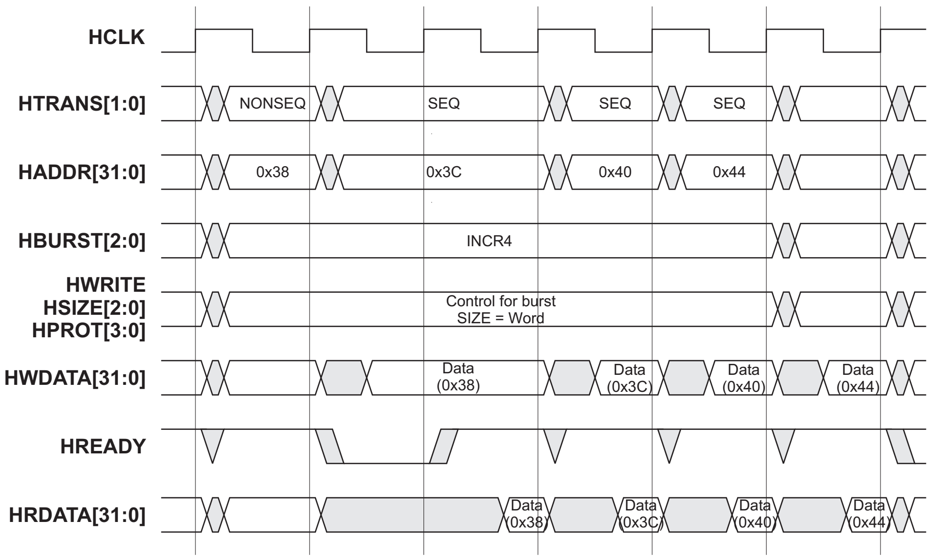


Figure 2.2.: Four beat incremental burst transfer on AHB bus [126]

Describing further details of the AHB protocol is out of the scope of this work and can be found in AMBA specifications [126].

2.2. Network on Chip

When the number of communicating nodes scale, shared-bus based architectures lead to performance penalties which make them unsuitable for multi and many core systems. Because of the communication bottlenecks of bus-based infrastructures, the real potential of the multiple processor systems could never be exploited. Therefore, the idea of realizing Network on Chip gained attention when the system designers had to search for the alternate communication infrastructure for next generation SoCs. Network on Chip are realized by keeping in view the concepts of conventional off-chip networks and applying them to on-chip architectures [7]. However, in comparison to conventional networks, NoC has distinctive characteristics like low communication latency and fewer power consumption [33].

Network on Chip provide a systematic approach to reuse communication resources in massively parallel SoC architectures. In addition, they enable integration of other architectural building blocks by providing a seamless communication interface [73]. A tiled architecture with NoC as communication backbone is represented in the figure 2.3, which shows how Network on Chip enable platform-based design approach for realizing many-core systems.

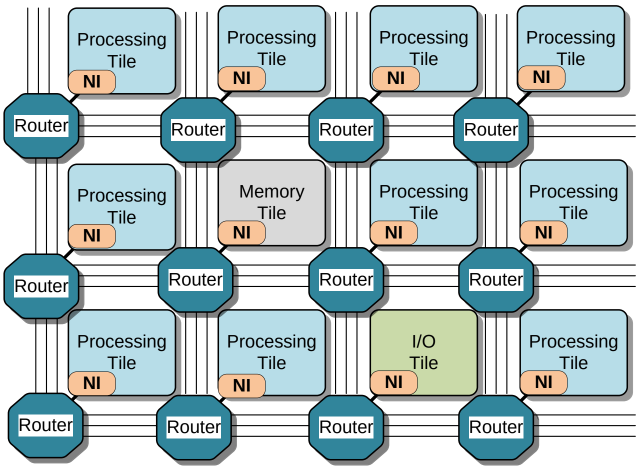


Figure 2.3.: NoC based tiled architecture

In the following text, a brief introduction about the Network on Chip and its basic components is provided for better understanding of our concept.

2.2.1. Major Components

As it is also clear from the figure 2.3, a Network on Chip generally consists of routers (also referred as switches), which are connected by the links to form a network according to a given network topology. The individual nodes, which are commonly referred to as tiles in so-called tiled architectures, are attached via network interface modules.

2.2.1.1. Router

A router is responsible for forwarding the data from an input port to the output port. In order to send the data from source to the destination node, the data transfer happens in multiple routers which are present in that path. Packet switching and circuit switching are two important methodologies for transferring data between routers. The details about these data transfer schemes will follow in section 2.2.3. The micro-architecture of a state of the art packet switching NoC router is shown in the figure 2.4.

Routers contain buffers which provide temporary storage to the input data until it can be forwarded to the output. In the figure 2.4, the buffers are placed at the input port of the router. In order to resolve congestion problem in the Network on Chip, virtual channels are used, which require multiple buffers per input port. The details about virtual channels in NoC are provided in the section 2.2.4. Routers define routing strategies to deliver the packet to the destination. Based on the routing decision, the output port can be allocated to forward the data. Routing unit in the router micro-architecture is responsible for

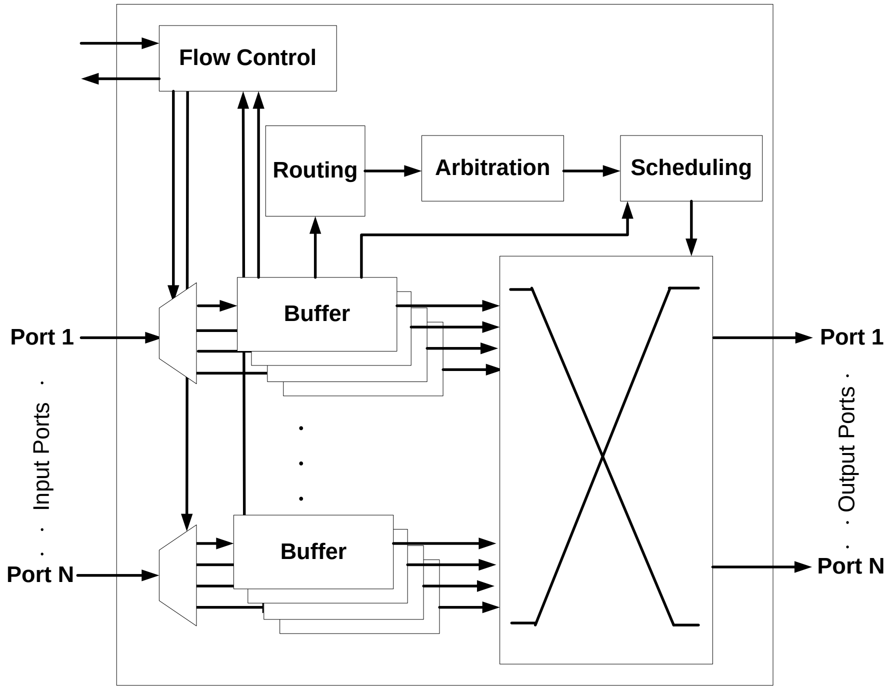


Figure 2.4.: Micro-architecture of a Network on Chip router [53]

performing the routing decisions. In order to ensure lossless transmission by keeping in view buffer fill status, flow control is incorporated. The data transmission at the output port is managed by the scheduling unit. If multiple input ports want to forward the data to the same output port, the selection of the input port is performed by the arbitration unit. In addition, the arbitration unit contains a reservation table to maintain the connection record between given input and output ports. The crossbar contains the multiplexer network which connects the input to the output ports according to the reservation table.

2.2.1.2. Network Interface

The network interface is the component which connects the nodes with the routers of the NoC. Network interface decouples computation from communication by offering an abstraction, which is independent of the network implementation [13]. Thus, the network interface is a protocol translator which maps the I/O protocol of the processing node or tile into the protocol used by the NoC and the other way round [10]. Essentially, each network interface has two interfaces: the NoC interface, which is used to connect with the Network on Chip router, and the tile interface, which is used to attach the components of the tile to the Network on Chip. The tile interface implements a standard point-to-point or bus protocol allowing IP reuse across different platforms. The use of existing bus protocols allows backward compatibility. The NoC interface of the NI is also responsible for packetization and routing related functions. The different interfaces of a typical network interface are shown in the figure 2.5.

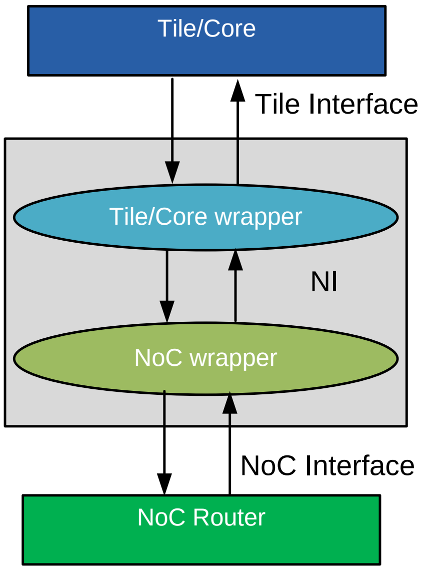


Figure 2.5.: Network interface ends

Network interface holds a very important position in NoC-based many-core architectures because it is directly connecting to both computation and communication domains. Besides playing the role of a protocol translator, network interface provides different hardware/software interfaces to the applications to communicate over the Network on Chip. In our concept, we have extended the role of network interface architecture to provide communication resource management and software communication support as additional services for improving system performance. The details of which will come in the chapter 3. In the following text, the important details about the Network on Chip are given, which are relevant for our concept.

2.2.2. Topology

The topology has an impact on performance and implementation cost of a Network on Chip. Implementation cost of each router is affected by the topology because it derives the number of ports in the router. In addition, the performance of the entire architecture is affected by the topology because it drives the bandwidth which can be supported by the network [17]. Figure 2.6 represents the major state of the art topologies for Network on Chip.

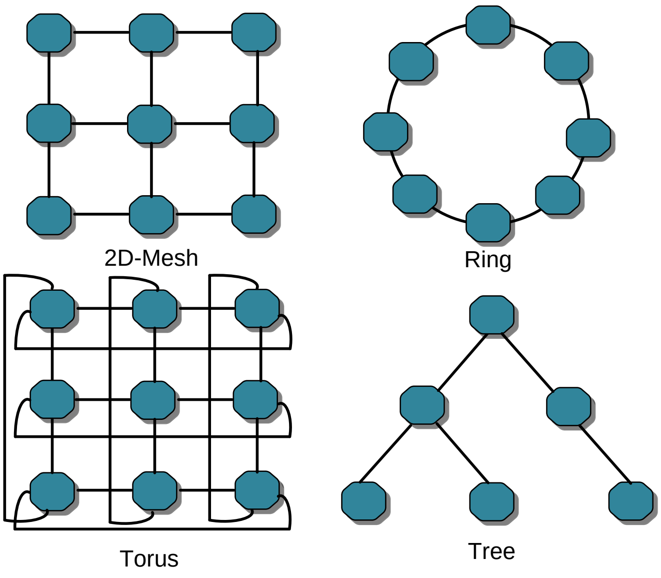


Figure 2.6.: Major Network on Chip topologies

The selection of the topology also affects the Network on Chip floor planning. Floor planning of tiled architectures with regular topologies is easier to realize and results in lesser average wire length as compared to irregular topologies. Hence, mesh topology is the most widely used Network on Chip topology. However, researchers have proposed irregular Network on Chip topologies which bring benefit for applications-specific architectures [105].

2.2.3. Switching Schemes

Switching technique is another important aspect which affects implementation cost and throughput of a Network on Chip based system. Switching methodology influences many design parameters of the Network on Chip. Switching mechanism defines the data flow over the network and granularity of data transfer. The minimum granularity of data which can be transferred over a Network on Chip is called flit. A flit represents the minimum datagram which is used for switching or flow control. In a packet switching based NoC, the communication happens in the form of packets. Several flits constitute a packet. For applications, the communication unit between two instances over Network on Chip is commonly referred as a message. A message may consist of one or more packets. There are two basic modes for data transfer within a network:

- **Circuit switching :** In circuit switching methodology, an end-to-end connection is established between transmitter and receiver before the actual data is sent [66]

- **Packet switching :** In packet switching concept, the data can be sent without prior connection setup. Packets may follow an independent route to the destination and hence the delay experienced by them may also be different. Because of the higher flexibility, packet switching is more widely deployed in Network on Chip.

In the scope of this work, we will discuss the details corresponding to packet switching scheme as this is the approach which we have deployed in our concept. Within packet switching, there are three important methodologies to address the data flow over Network on Chip:

1. **Store and Forward :** In this approach, a packet is only forwarded from one router to the next, if there is enough space available in the buffers of next router to store the complete packet. This approach has an advantage that the packet transmission between two routers never stops when it is started. This is due to the fact that it is ensured that all flits within the packet would be accepted by the receiver. However, the concept comes with the disadvantages of high overall delay and large buffer size requirement.
2. **Virtual Cut Through :** In comparison to store and forward approach, this methodology does not require that the complete packet should be stored in one router in order to wait for the readiness of the next router. If there is not enough space available in the next router, the whole packet has to be buffered. This approach offers the advantage of reduced delay as compared to store and forward approach. However, the disadvantage of large buffer size requirement still exist.
3. **Wormhole :** In this concept, the packet must not be completely stored in the router buffer before it can be forwarded to the next router. Therefore, the receiving router does not need to have buffer space for storing the entire packet. This approach offers the advantages of both lesser buffer size and reduced latency as compared to the other two approaches. However, the disadvantage is the possible blocking of links by long packets which span through multiple routers.

As the buffer size is the most important consideration in the Network on Chip design, wormhole switching is the most widely used switching methodology. In order to address, the link blocking problem caused by wormhole switching, the concept of virtual channels is introduced in NoCs.

2.2.4. Virtual Channels

The concept of virtual channels (VC) was already introduced for communication networks by Dally in 1992 [27]. Kavaldjiev et al. brought the concept of virtual channels in the Network on Chip architectures [80]. A network interface architecture with VC based design was presented by Bhojwani et al. [11]. Virtual channels enable the sharing of the physical link by multiple traffic flows. In this way, the performance is improved by increasing the overall link utilization [97]. In addition, virtual channels help in resolving deadlocks over Network on Chip. The details about the deadlocks and the ways to resolve them would be provided in section 2.2.8. However, virtual channels bring additional implementation cost as well as the complexity, which includes the scheduling requirements of virtual channels over the link.

2.2.5. Scheduling and Arbitration

Virtual channels are required to be scheduled for data transmission over the link. The scheduling may also be referred as arbitration. In the following, the most relevant arbitration schemes for Network on Chip are given:

- **Time Division Multiplexing (TDM) :** This scheme uses fixed size time slots which are statically assigned to each traffic flow corresponding to a virtual channel. The advantage of this approach is the less implementation cost and guaranteed throughput for each traffic flow. The drawback of the approach is the reduced throughput because of the scheduling of idle slots. The scheduling of the idle slots results in an inefficient utilization of the available bandwidth.
- **Priority :** Priority based scheduling policies define precedence of certain traffic flow(s) over other traffic streams while scheduling them on the physical link [31]. The drawback of priority scheduling is the starvation of traffic flows with low priorities by the ones with higher priority.
- **Round Robin :** Round Robin is an arbitration scheme which ensures fairness in terms of share of each traffic flow over the link as compared to priority based scheduling. Compared to TDM, round robin arbitration does not take into account the idle slots for scheduling. Not considering the idle slot, results in high implementation cost for the arbiter. The efficient link utilization and fairness make round robin a widely used scheduling strategy for Network on Chip.
- **Weighted Round Robin :** Weighted Round Robin (WRR) scheduling is a well-established scheduling concept in ATM switches [77]. It is the extension of round robin scheduling. Like round robin scheduling, it serves all traffic flows within one scheduling cycle. However, the number of schedules per cycle for each traffic flow can be different and depends on the weight which is assigned to the respective flow. The length of a scheduling cycle is the sum of the assigned weights for all traffic flows as described by the equation 2.1.

  $$WRR\_Cycle\_sch = \sum TF\_Timeslots_i \tag{2.1}$$

  Where WRR_Cycle_sch represents the scheduling cycle of weighted round robin scheduling and $TF\_Timeslots_i$ defines the number of time slots, which are assigned to the traffic flow i. The WRR arbitration policy is illustrated in the figure 2.7. With the possibility of assigning multiple time slots to a single traffic flow, Quality of Service requirements can be achieved for individual applications.

2.2.6. Flow Control

Flow control is required over Network on Chip to ensure lossless communication between transmitter and receiver. In Network on Chip, data can be lost because of two reasons; 1) issues during the data transmission like the unavailability of buffer space in the receiver, 2) data being overwritten due to lack of synchronization between transmitter and receiver. Therefore, flow control is required to be addressed at multiple layers in Network on Chip.

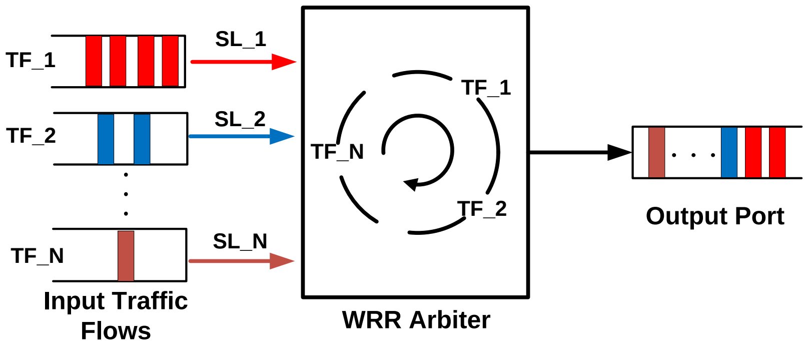


Figure 2.7.: Weighted round robin arbitration policy

The flow control which is required at the link level between the neighboring routers or between network interface and router is referred as link-level flow control. Whereas the end-to-end flow control deals with the communication between transmitting and receiving nodes. However, we will discuss only link-level flow control in this section. This is due to the fact that in our work, the end-to-end flow control is managed by the application layer. Important state of the art flow control mechanisms used in Network on Chip are following:

- **Credit-based flow control :** In this flow control methodology, the sender is allowed to pro-actively transmit a certain amount of data without waiting for explicit acknowledgment from the receiver. This specific amount of data is termed as credit, which corresponds to the buffer space available in the receiver. Each time a flit is transmitted, the credit is decremented in the transmitter. If the credit becomes zero, it indicates the unavailability of space in receiver's buffer. In this case, the transmitter is not allowed to keep sending data and must wait until the credit becomes available again. When the receiver has space available in its buffer, it signals back to the transmitter. When the transmitter receives this signal, the credit is incremented and the transmission can be continued.
- **Acknowledge-based flow control :** This protocol is based on explicit acknowledgment before the data transmission which is indicated through signals/wires between transmitter and receiver. Therefore, the transmitter waits for the readiness of the receiver before sending any data. The sender indicates the willingness to transmit the data through a signal. When the receiver is able to accept the data, it responds with an acknowledge signal. If the transmitter does not receive the acknowledgment, the flits have to wait in the buffer of the transmitter till the time the receiver is ready to accept them.

2.2.7. Routing Mechanisms

Routing is the process by which a packet finds its path from the source to the destination node. The routing mechanism strongly depends on the Network on Chip topology. Choice of routing algorithm affects many aspects associated with the Network on Chip design like power consumption, resource requirement, performance and deadlock resolution. The important routing strategies in Network on Chip are given as follows:

- **Source Routing :** In this routing scheme, the transmitter defines the complete route of the packet from source to the destination [96]. The routing information is encoded in the packet header by the network interface which is then processed by each router in the path and the packet is forwarded accordingly. The routing is done with the help of a static routing table placed within the routers. Because the routing is performed in the source node, the complexity of the Network on Chip routers can be reduced. The drawbacks of source routing are overhead of encoding complete route in the header of each packet and the lack of scalability because of the static route selection at the transmitter.
- **Deterministic Routing :** Deterministic routing uses static paths between the source and the destination pair. However, as compared to the source routing, there is no need to encode complete routing information in the packet header at the transmitter. Each router in the path takes routing decision according to the defined routing methodology. In comparison to the source routing, the deterministic routing is beneficial as it avoids the need for large look-up tables in the network interface and saves the route encoding overhead. The disadvantage of this approach is a small degree of freedom for path selection which makes it less flexible in dynamic load conditions. However, the deterministic routing offers low-cost implementation as compared to the adaptive routing techniques. Considering above-mentioned facts, deterministic routing schemes like XY routing are the most widely used routing strategies in Network on Chip.
- **Adaptive Routing :** This routing algorithm can provide different paths between the given source and destination node pair. Within the list of possible routes, the most suitable path can be selected for transmission at run-time. The advantage of the approach is the optimum route selection keeping in view dynamic load, power consumption, and fault tolerance [69]. However, the adaptive routing comes up with additional implementation cost and may lead to deadlocks in Network on Chip.

2.2.8. Deadlocks

In Network on Chip, a deadlock occurs when packets in the network are blocked because they are waiting for an event which can never happen. There are two types of deadlocks which may occur in Network on Chip:

- **Routing-dependent deadlock :** In this category, the deadlocks are included which occur because of the inappropriate routing algorithm. Only Network on Chip routers are involved in these deadlocks. Especially, wormhole switching is susceptible to deadlocks because the packets are spread over multiple hops. The solution to these types of deadlocks is to choose a routing algorithm which is deadlock free. The basic principle of having deadlock-free routing algorithm is to prohibit certain turns [68]. For example, XY routing avoids certain turns and hence is a deadlock free routing mechanism. Virtual channels are another way of resolving routing-dependent deadlocks.

- **Message-dependent deadlock :** Message-dependent deadlocks may occur even if Network on Chip operates with a routing algorithm which is deadlock free. Message-dependent deadlocks arise because of the interdependency of different message types, for example, request and response message dependency. These different types of messages represent an abstraction through which different nodes communicate over the Network on Chip. An example of a message-dependent deadlock is shown in the figure 2.8 where the communication between compute and memory tiles is suspended because of read/write request and response message dependency. Message-dependent deadlocks can be resolved by using a separate virtual channel for each message type in the Network on Chip. Many advanced approaches have been proposed to resolve message-dependent deadlocks [86] [99].

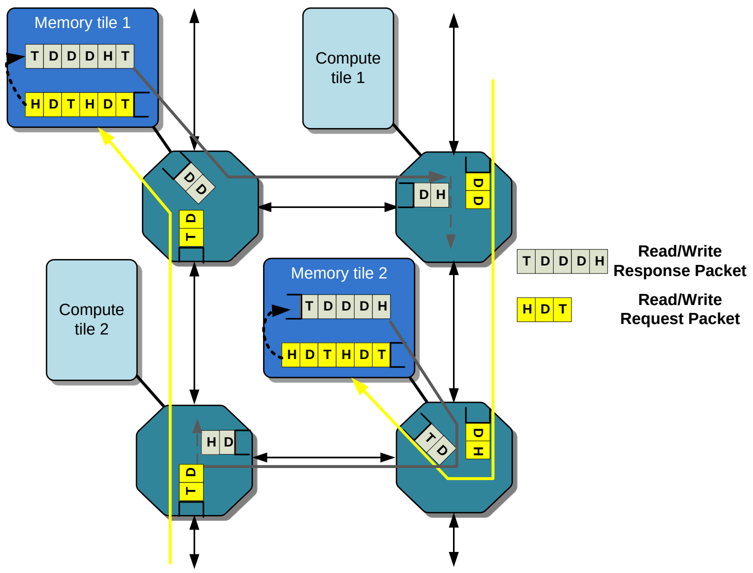


Figure 2.8.: Message-dependent deadlock example

2.2.9. State of the art Network on Chip Architectures

In this section, we review and discuss state of the art Network on Chip implementations [1]. Since our proposed concept is based on the network interface architecture, therefore, we will focus more on the details which are related to the network interface design.

2.2.9.1. Æthereal

Æthereal NoC developed by Phillips aims at achieving predictability in communication infrastructure design. Therefore, guaranteed throughput and latency is supported over Network on Chip to provide deterministic communication behavior [48] [121]. Æthereal Network on Chip supports configuration of important parameters at run-time. Wormhole routing with input port buffers is used to route the flits. In addition, it supports both Best Effort (BE) and Guaranteed Throughput (GT) communication services [49]. A time-division multiplexed circuit switching approach is employed for guaranteed throughput communication.

Network interface implemented in Æthereal considers only shared-memory programming model support [118]. Memory read and write operations are supported through a transaction-based protocol. NI offers the standard bus interfaces like AXI and OCP to connect other IP modules. The network interface architecture consists of two parts; the NI kernel and the NI shell. NI kernel handles the Network on Chip communication-related aspects like realizing the channels, preparing packets and scheduling them over the link. In addition, it is responsible for end-to-end flow control between the communicating nodes. NI shell is responsible for setting up narrowcast and multicast connections, transaction ordering, and other high-level aspects, which are related to the protocol offered to the applications. The NI kernel communicates with the NI shell via ports. Network interface developed by Æthereal concedes higher latency as compared to the other state of the art network interface architectures because of its generic implementation. In addition, it does not address shared memory synchronization support, which is essential for applications when they are executed over Network on Chip based architectures.

2.2.9.2. Xpipes

Xpipes is the Network on Chip architecture targeted at high-performance and reliable communication for embedded multiprocessors [8] [26]. Xpipes Compiler is introduced as a framework for automatically instantiating customized NoC components (switches, network interfaces, and links) from the SystemC-based implementation. A static routing protocol called ”street sign” routing along with wormhole switching are incorporated. Static routing information is accessed by the header builder, which encodes this information into the network packet header. Xpipes implements an error control logic, which retransmits the data packets which are not acknowledged.

The Xpipes network interface architecture supports shared memory accesses. The proposed architecture supports multiple outstanding write transactions but a single outstanding read transaction to avoid deadlocks. However, the network interface of Xpipes does not support more advanced communication features in order to reduce the system complexity. It provides the standard OCP interface to the processing cores [37]. Network interface sharing based methodologies are introduced which allow the network interface to be shared by multiple nodes and hence result in less area consumption. For NI sharing, dedicated hardware units are introduced in the architecture to merge and split different traffic flows on a single network interface.

2.2.9.3. Nostrum NoC

Nostrum NoC is a Network on Chip proposed to use packet switched communication with the store and forward switching mechanism [94]. It supports both Best Effort and Guaranteed Bandwidth (GB) traffic flows. In Nostrum NoC, guaranteed bandwidth traffic class is supported by a concept called looped containers [93]. Looped containers are realized by virtual circuits. These virtual circuits use an explicit time division multiplexing mechanism, which is named as Temporally Disjoint Networks (TDN).

In Nostrum NoC, network interface functionality is divided in Network Interface (NI) and Resource Network Interface (RNI) modules [125]. RNI manages the interface with the processing cores whereas NI translates the incoming requests from processors according to Nostrum protocol. RNI provides a wrapper functionality to AMBA AHB bus protocol. Read and write to the memories are supported [2]. The Nostrum NoC does not address the hardware support for software synchronization.

2.2.9.4. Mango

Mango is a clock-less Network on Chip which supports both guaranteed service and best effort communication [15]. It uses a chain of virtual channels to establish virtual end-to-end connections. These virtual end-to-end connections are the means to provide Quality of Service. Router architecture is divided into two parts, which deal with guaranteed service and best effort communication respectively. The Mango Network on Chip uses wormhole switching and XY routing.

Network interface architecture by Mango NoC developers is OCP compliant [14]. NI architecture consists of Core Interface (CI) and Network Interface (NI) components, which establish the interface to processing cores and Network on Chip router respectively. The synchronization between the clocked processing core and the clock-less network is also performed by the network interface. In addition to shared memory accesses, the network interface supports interrupts based on virtual wires. Interrupts are sent from source to the destination node as special packets. On receiving the special packet, interrupt is raised at the Core Interface end so that it can be processed by the respective processing core.

2.2.9.5. SCORPIO

The SCORPIO architecture is based on Network on Chip architecture with cache coherence hardware support [32]. Cache coherence is supported through a snooping based protocol which relies on distributed message ordering instead of global message ordering. The Network on Chip consists of two physical networks to separate message ordering from message delivery. The main NoC is an unordered network, which is responsible for sending and receiving of the coherence requests. For each coherence message which is dispatched on the main network, a notification message is sent on a separate NoC, called notification network. The notification Network on Chip is buffer-less fixed latency network to ensure in order delivery of notification messages.

Network interface serves both main and notification network routers. At the sending end, it injects the data messages and the coherence requests into the main network. The corresponding notifications are sent through the notification Network on Chip. On the receiving side, the network interface calculates the global order of the cache coherence sequence locally according to a consistent ordering rule and then forward these requests to the cache controller. Therefore, the requests may arrive in any order but they are served at the network interface of each node in the same order.

After discussing the state of the art Network on Chip realizations and in particular the network interface designs, the role of the network interface in Network on Chip based systems is characterized further in the following text.

2.3. Conventional Communication Support in Network Interface

The details about the Network on Chip components and the state of the art NoC implementation highlight that the services offered by the network interface design are defined to meet the following objectives:

- Decouple computation (Processing cores) and communication (Network on Chip)
- Provide backward compatibility to tile interconnect protocol
- Hardware support for efficient communication and synchronization

First two design objectives are achieved through tile and network protocol translation, which are detailed in the following section 2.3.1. Aspects specific to communication and synchronization support are discussed in sections 2.3.2,2.3.3 and 2.3.4 respectively.

2.3.1. Tile and Network Protocol Translation

Transparent translation of the tile interconnect and NoC protocols is the prime responsibility of the network interface. Bhojwani et al. [10] proposed following three strategies to perform tile and network protocol translation:

- **Software realization :** The software realization of the protocol translation requires the implementation of certain library functions. These library functions consist of special commands/instructions, which are executed when the data is required to be sent over the Network on Chip. This approach also requires extensions in the Instruction Set Architecture (ISA) of the processor which initiates these requests.
- **Processor micro-architecture enhancement :** This approach requires a custom module within the processor micro-architecture to trigger the IP interconnect to NoC protocol translation. This module is customized according to the processor interface and the Network on Chip router, which are deployed in that system.
- **Protocol translation wrapper :** The above-mentioned two strategies are less flexible keeping in view the required instruction set adaptations or the micro-architecture enhancements. Therefore, incorporating the network interface in the form of a wrapper to interface the processing cores with the Network on Chip is the widely

used approach. This wrapper provides the functionality of protocol translation, packetization, and de-packetizing of the NoC transactions.

The wrapper-based approach completely separates the computation and communication domains and does not require custom extensions in the architecture of either the processor or the Network on Chip. Protocol translation through wrapper-based approach requires wrapper development for both tile interconnect and Network on Chip interfaces. Both tile bus and NoC wrapper provide services corresponding to the transport layer in the ISO-OSI reference model [150]. The details about these services are given in the following sections.

2.3.1.1. Tile bus Wrapper

Tile bus wrapper can be considered as an implementation of the session layer in the OSI network model. It transforms bus protocol transactions into packets corresponding to Network on Chip protocol. Bus transactions include both request and response transfers. The requests arriving at the tile bus are issued by the bus masters like processing cores and hardware accelerators. Response transfers are issued by the bus slaves like memory in reply to the bus requests. Protocol translation and the packetization are kept transparent by the tile bus wrapper from the bus masters and slaves. Backward compatibility is achieved by customizing the tile bus wrapper for the state of the art bus standards. Researchers have addressed the tile bus wrapper functionality of the network interface for several on-chip bus protocols. Ebrahimi et al. have proposed a network interface architecture with AMBA AXI wrapper [34]. Attia et al. have developed a low latency bus wrapper for AMBA AHB protocol [3]. A socket based approach to provide wrapper interface for two different bus protocols i.e. AMBA AXI and STBus is provided by Mereu et al. [90]. Olsen et al. have presented an OCP wrapper functionality [108].

2.3.1.2. NoC Wrapper

Network on Chip wrapper translates the incoming requests from NoC, which are arriving in the form of packets, according to the bus protocol. NoC wrapper is designed to meet the Network on Chip protocol requirements. Chang et al. proposed a Network on Chip wrapper for custom NoC protocol [21]. Jung et al. present a network interface with NoC wrapper for a custom on-chip network, SONA [76].

2.3.2. Remote Memory Access

The network interface supports mechanisms to enable real-time communication between remote tiles. From the software perspective, the communication mechanisms could be shared memory accesses or data exchange in the form of message passing. These different ways of communication are handled by the software layers according to the corresponding programming model. Memory communication is the most significant communication mechanism supported by the network interface keeping in view the shared memory programming model. Especially, it is important to access application data and code which are placed in either on-chip or off-chip memory. Network interface enables access to the tile external memory through implicit load/store based shared memory communication as well as through efficient data transfer mechanisms like remote direct memory access (RDMA).

2.3.2.1. Remote Shared Memory Access

Remote shared memory access support consists of provisioning load and store accesses between tiles over Network on Chip. Remote load/stores represent shared memory abstraction primitives which enable transparent access to the external memory. These primitives enable the use of legacy application code without caring about the underlying architecture. If the external memories of other tiles or the global memory are mapped into the address range of the network interface, they can be accessed transparently through remote load/store requests. From the tile bus perspective, load/stores are transactions which are blocking in nature. Therefore, bus master must receive the response for current load/store request in order to proceed with the next bus transaction. Transactions between master and slave components which cover shared memory accesses are represented in the figure. Remote load/store requests are realized using request and response messages over Network on Chip.

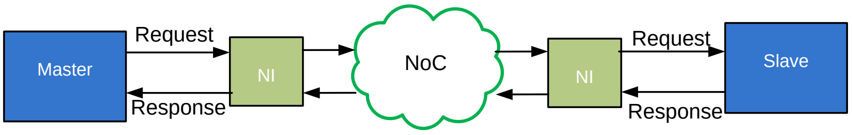


Figure 2.9.: Transactions representing interaction between master and slave for shared memory access

Kavadias et al. have presented a network interface design which provides access mechanisms to both centralized and distributed shared memory [78]. In [78], shared memory accesses between different tiles are cache coherent. Cache coherence is supported by providing L2 caches within the network interface architecture.

2.3.2.2. Remote Direct Memory Access

Remote direct memory access is the widely used approach for efficient data transfer between memories. Hardware engines are developed which allow direct memory to memory communication. Direct transfers are expected to provide both performance and energy advantages when compared with the load/store based accesses. In addition, RDMA could be realized in the form of an asynchronous non-blocking protocol compared to blocking load/store transactions.

In order to initiate an RDMA operation, software must pass certain arguments to the hardware engine. Lee et al. present network interface with register interface for providing DMA support to the applications [87]. In this work, the register interface is accessible by OpenRISC processor [85]. In order to provision distributed memory programming, direct memory access is supported for data copy operations between remote memories as proposed by Kavadias et al. [78].

2.3.3. Quality of Service Support

The Quality of Service support for conventional networks is an established concept. According to QoS concepts for conventional networks, different traffic flows are served with their desired priority level or so-called differentiated service classes [124]. NoC based multiprocessor systems facilitate execution of concurrent applications over the underlying architecture, which may have characteristic throughput and bandwidth requirements. Therefore, it is crucial to support Quality of Service at NoC level subsequently in the network interface as well. Since the introduction of Network on Chip, researchers have investigated concepts to provide QoS guarantees [36].

Bolotin et al. have proposed a customized architecture, where different application classes with different QoS requirements are served. The Quality of Service requirements are met by using dedicated communication resources, which are allocated for the respective applications [16]. Virtual channels are the communication resources, which are available in the Network on Chip and could be explicitly managed to support the traffic requirements of applications. End-to-end virtual channel reservation offers a simple and flexible concept for delivering Quality of Service in a packet switched NoC [79].

2.3.3.1. Guaranteed Service and Best Effort Traffic

Guaranteed service and best effort represent two broad traffic classes which are available to applications. The characteristics of the two traffic types can be detailed as follows:

- **Best Effort :** BE traffic does not make any commitment regarding Quality of Service. It refers to basic connectionless traffic without guarantees [75]. Best effort communication is essential where the number of communication partners is not limited e.g. multicast and broadcast communication.
- **Guaranteed Service :** Guaranteed service traffic ensures predetermined bandwidth and latency to the applications, which require explicit QoS guarantees regardless of other traffic over the network. GS traffic is supported through end-to-end connections which are realized either using TDM-based mechanisms or circuit-switching. In the scenarios where the applications communicate with a limited number of partners, GS connections reduce packet switching overhead for data transmission and hence save the power consumption.

There are multiple ways in which GS and BE traffic flows are initiated and supported by Network on Chip router and network interface. Rijpkema et al. have proposed a Network on Chip architecture, where GS and BE traffic flows are handled within the same router architecture [121]. Coexistence of GS and BE in a virtual channel based Network on Chip is shown in the figure 2.10. In a virtual channel based NoC, GS connection is initiated by issuing the Connection Header (CH) from source to the destination tile. The connection header performs end-to-end virtual channel reservation. Afterward, the reserved virtual channels are used by the data traffic between source and destination tiles. Best effort communication does not require end-to-end virtual channel reservation.

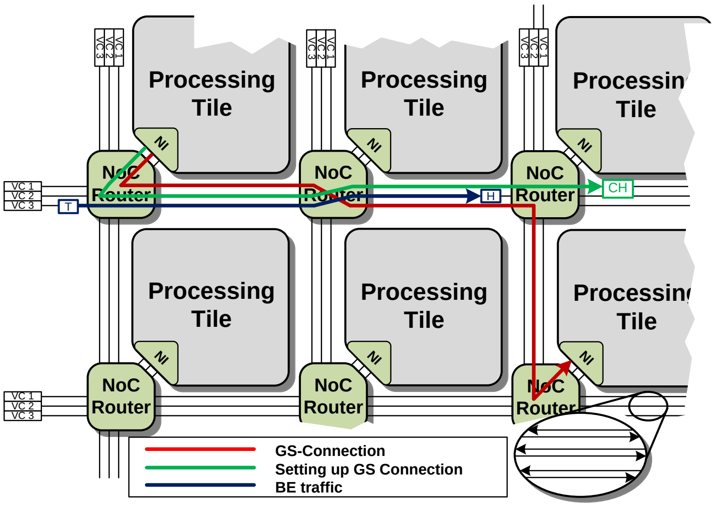


Figure 2.10.: GS and BE traffic flows over Network on Chip

2.3.3.2. Communication Resource Management

In Network on Chip, virtual channels are the shared communication resources which are utilized by the concurrent applications. The applications with best effort communication do not explicitly control the allocation of communication resources. Whereas in the case of applications with QoS requirements, the allocation of communication resources is directly influenced by those applications. In state of the art approaches, the application or the operating system has to be aware of the communication requirements of applications before their execution on underlying platform [20]. An operating system or high layer software instance initiates the request for guaranteed service communication on behalf of applications. According to the traffic requirements, NoC router and the network interface are capable of reserving virtual channels for offering throughput guarantees to the corresponding applications. In addition, the role of the network interface is to provide a configurable interface, which could be used by the applications for initiating and releasing the reservation of communication resources.

The need of an efficient communication resource management approach in a virtual channel based NoC is depicted through a scenario in the figure 2.11. In the shown scenario, the application instances A_1, A_2, A_3 and A_4 communicate via best effort traffic. Similarly, application instances C_1, C_2 and C_3 also use best effort communication. Guaranteed service communication is deployed between application instances B_1 and B_2. In the figure 2.11, software instance B_1 initiates the reservation of communication resources between B_1 and B_2 before the start of actual data transfer. In this scenario, the communication resource allocation does not consider the application mapping on the underlying platform. Therefore, the GS connection is established between two distant nodes i.e. B_1 and B_2, which requires reservation of a large number of virtual channels. This uncoordinated resource assignment influences the performance of other concurrent applications and results in the sub-optimal utilization of communication infrastructure. Hence, an appropriate communication resource allocation scheme is required, which considers dynamic factors like application mapping and underlying platform status while performing resource management. .

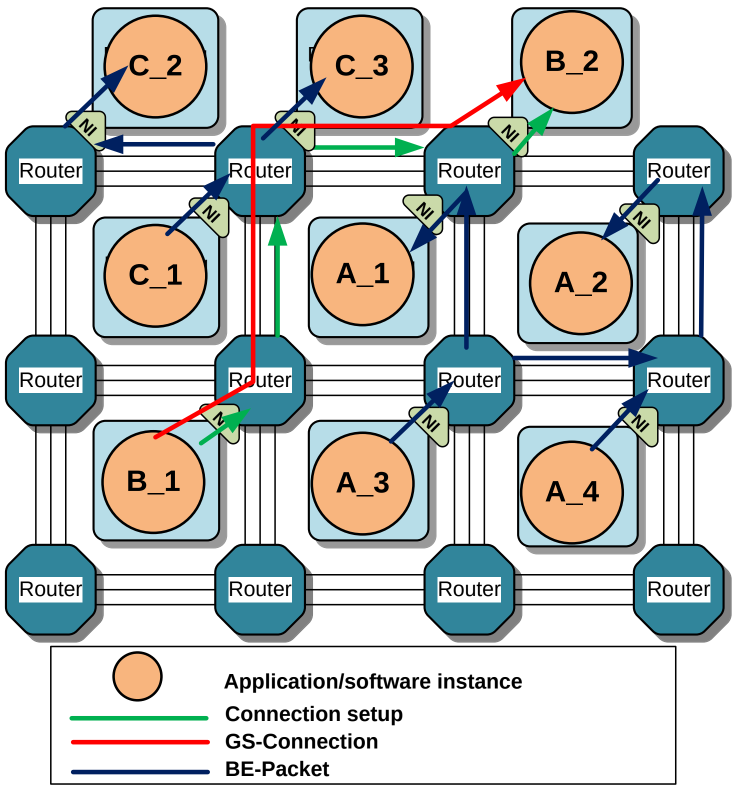


Figure 2.11.: Network on Chip communication resource management

Researchers have followed different approaches to develop system level communication resource management strategies. Bin et al. have advocated for a coordination based approach for distributing Network on Chip resources among applications. In that work, the instance which coordinates the management of resources is implemented in software [88]. Software based communication resource allocation relies on the static knowledge about the application's communication characteristics. These mechanisms are unable to adapt themselves according to run-time application mapping and communication bandwidth requirements. On the other hand, hardware-based approaches advocate the need of a hardware unit, which performs the management of communication resources in a centralized manner [51], [107], [83]. Centralized solutions are not scalable keeping in view the run-time communication behavior of applications on distributed shared memory architectures. Keeping in view above-mentioned considerations, we have proposed a distributed hardware-based scheme to allocate communication resources. The details of our approach will follow in chapter 3.

2.3.4. Inter-tile synchronization support for software layers

In order to exploit task level parallelism, it is important that the software layers which are being executed on different tiles, are able to synchronize without significant performance overhead. In initial attempts, the system developers used remote memory access support in the network interface for synchronization. However, keeping in view the communication and synchronization demands of real world applications, the remote memory access support was found to be inappropriate for this job. Increasing architecture sizes have encouraged the development of novel methodologies to provide synchronization support for software layers between remote nodes. Tota et al. have proposed an architecture with synchronization support, which is based on a custom hybrid shared memory/message passing approach [132]. In the proposed hybrid communication model, the data exchange happens via shared memory whereas the synchronization is done through message passing. For message passing, basic synchronization primitives MPI_send(), MPI_receive() and MPI_barrier() are implemented, which support direct processor to processor communication. Synchronization for shared memory is not addressed in this concept. Different components of the hardware architecture are modified to support synchronization. Processor pipeline is extended to support custom instructions for shared memory access and message passing synchronization. Moreover, the interface between processor and network interface is adapted to the proposed hybrid shared memory/ message passing communication model. The work does not present the comparison with the other state of the art approaches. In addition, it is not investigated that how the proposed hardware enhancements can be beneficial for state of the art programming models.

Chen et al. have proposed a hybrid hardware-software solution to support a distributed shared memory architecture [23]. The authors present a Dual Microcoded Controller (DMC) module which contains multiple sub-modules for supporting operations like core and network interfacing, virtual to physical address translation and synchronization. Synchronization between remote network nodes is supported through test-and-set primitive. The benchmarks which are used as test cases for supporting shared memory and message passing execution are quite limited in scope. In addition, it is not addressed that how this approach can be applied to real-world applications. Kavadias et al. presented a hardware-software based approach to provide communication and synchronization interface for software layers [78]. In this approach, direct memory access accelerator is used for data transfer between nodes. According to this concept, software is performing and controlling the synchronized accesses which are happening between remote tiles over Network on Chip. Synchronization is done through state of the art locks and barriers.

The synchronization overhead over distributed interconnect while using software-based synchronization is not addressed.

Above-mentioned hardware oriented approaches target customized programming models which limit their applicability to wide range of applications supporting state of the art shared memory/message passing execution. Software controlled synchronization approaches have a disadvantage that the software has to poll the status of each respective sub-operation in order to perform the next subsequent operation. Status polling results in significant overhead on tile interconnect. In addition, it limits the system performance as the software remains busy with the status polling and can not proceed with the actual application processing. None of the above-mentioned approaches focus on the synchronization support related to task spawning, which plays a vital role in exploiting the task level parallelism of the underlying architecture. To the best of our knowledge, there is no existing approach which addresses communication support for all stages of task spawning operation. In our work, we have focused on the synchronization support for task spawning. In addition, our investigations are not restricted to a particular programming model. The details of our concept for communication resource management and synchronization support will follow in chapter 3.

2.4. Invasive Computing

In general, the concepts presented in this work are focused on the next generation distributed shared memory architectures. However, the proposals have been investigated under the project named Invasive Computing, which refers to a novel paradigm for designing and programming future parallel computing systems. The invasive computing paradigm is investigated in a Transregional Collaborative Research Center (TCRC) funded by the Deutsche Forschungsgemeinschaft (DFG). The following text provides a brief introduction to the basic concepts and the hardware architecture of the invasive computing project.

2.4.1. Concept

Invasive computing is motivated by the trend for designing next generation many-core architectures [131]. As it was pointed out in the introduction chapter that the number of processing cores will increase in future architectures. Keeping in view this forecast, the programming of architectures with a large number of processing cores gains importance. Invasive computing tries to address this challenge by using resource-aware programming. In past, concepts have been proposed which advocate the necessity of resource-aware programming in parallel architectures [89]. However, the novelty of invasive computing is to introduce resource-awareness in all system layers i.e. from the application to the architecture. Invasive computing advocates the need for bargaining based resource management between applications. Applications have the ability to dynamically adapt their computation taking into account their computational requirements and the available hardware resources. The concept of invasive computing is shown in the figure 2.12.

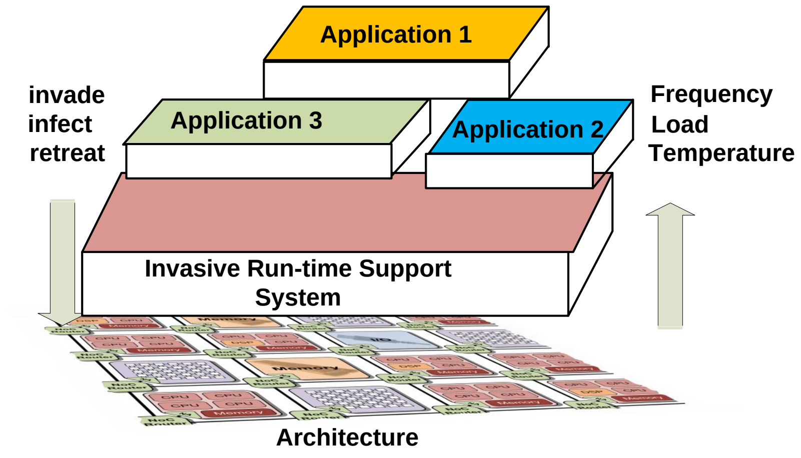


Figure 2.12.: The concept of invasive computing

In invasive computing, the execution of an application can be partitioned in multiple phases. Application requests for the resources in a phase named *invade*. The resource requirements of the application are expressed in the form of constraints. The constraints describe the quantity, properties, and type of resources desired by the application. For example, the application requirements could relate to the processing frequency, load or the temperature of processing cores. A higher layer software is responsible for finding resources which match the application requirements. The Invasive Run-time Support system is the name chosen for this software layer. Run-time support system takes into account the status of the underlying hardware for acquiring the suitable resources for the application. If appropriate resources are available, they are provided to the application as a *claim*. A claim is a set of resources which are temporarily reserved for the application. After these resources are allocated to the application, they are used by the application for its computation.

The phase in which the application starts its execution on the acquired resources is called *infect*. In the context of invasive computing, the unit of the workload which is executed on the underlying processing core is named as an *i-let*. Once the application completes its execution or the degree of parallelism reduces, the application enters into a phase called *retreat*. In this phase, the acquired resources are either completely or partially released. It is clear from the above description that the concept of invasive computing requires enhancements in the conventional application as well as architecture layers.

An invasive application has to be structured in a way that it can express its execution requirements in terms of desired platform resources. X10 language was adapted according to the principles of invasive computing. Originally, X10 is a programming language from IBM to program massively parallel architectures by using partitioned global address space programming model [22]. The extended version of X10 for invasive computing is named InvadeX10 [52]. InvadeX10 supports three basic invasive programming primitives i.e. invade, infect and retreat. In addition, it provides means to express the resource requirements for invasion by the use of the constraints. From the architecture perspective, the hardware resources should be modified in order to support temporal reservation for applications. In addition, the architecture should provide its current status to the higher software layers for performing resource-aware decisions.

Software and hardware modifications require a compiler which generates the executable for the invasive application to run on the underlying architecture. The front-end of existing X10 compiler was extended to support the invasive concepts and the constraint system. The compiler front-end was also modified to support the code generation for applications, which are written in C language. In addition to the changes in the front-end compiler, the back-end was implemented to support code generation for the invasive architecture [19]. Further details of the above-mentioned contributions are out of the scope of this work.

Novel simulation and design space exploration strategies to investigate several aspects related to software and hardware design in invasive computing were incorporated [146], [47]. An overview of the components involved in both software and hardware layers of invasive computing is provided in the following sections.

2.4.2. Software

In invasive computing, the modified application organization and the architectural extensions require support from the software layers. The software layer for invasive computing, which incorporates support for resource-awareness, is named Invasive Run-time Support System (iRTSS). iRTSS represents the distributed software layer, which is highly scalable and suitable for resource-awareness support in software. iRTSS further consists of the OctoPOS, the agent system, and hardware abstraction layer. The layered model showing different components of iRTSS and their positioning between application and architecture is shown in the figure 2.13. The OctoPOS is the name chosen for the operating system developed for invasive computing. The agent system refers to the distributed resource management layer whereas the hardware abstraction layer refers the interface to the different hardware back-ends of invasive computing. The details of the important components of iRTSS are described in the following text.

2.4.2.1. Agent System

The resource bargaining in the invasive computing is managed by a special component of the iRTSS, referred as agent system [82]. Agent system is a resource management system implemented in a distributed manner to enable scalability. In recent times, resource management methodologies which focus on resource bargaining are getting importance for many-core architectures [112]. The agent system for invasive computing consists of multiple agent instances. Each agent instance is either responsible for the resources within an architecture region or for the resources belonging to an application. Agent instances communicate with each other for bargaining resources. This approach distributes the resource management overhead over the entire system to meet the scalability requirements of next generation many-core architectures.

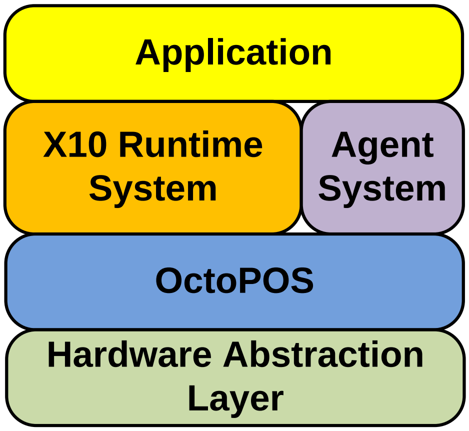


Figure 2.13.: Different layers of Invasive Run-time Support System

The agent system plays an important role during the invade phase. An invade request is handed over from the invasive application to an instance of the agent system. This request includes the constraints of the application for hardware resources. The agent instance is now responsible for searching and allocating hardware resources, which suit the application constraints. Therefore, the agent instance communicates with other agent instances in the system and takes the status of the underlying platform into account. The interfaces for communication and accessing status information are provided by the operating system. When suitable resources are found, they are reserved for the requesting application. Finally, the agent instance returns the set of available resources as a claim to the application.

2.4.2.2. Operating System

The operating system in the invasive run-time support system is named as OctoPOS. It connects higher software layers (applications, X10 run-time system, and agent system) and the invasive architecture [104]. Invasive applications execute directly on the OctoPOS operating system. One instance of this operating system is running on a particular core within each tile of the architecture. The individual OctoPOS instances communicate with each other to accomplish the functionality of a distributed operating system.

OctoPOS supports different types of hardware platforms. It can be executed on the SPARC V8 architecture, an x86 architecture or as a user process within a Linux based system. Keeping in view the characteristics of the invasive hardware architecture, OctoPOS has to support additional hardware features. In addition, OctoPOS provides the library of drivers for accessing all hardware components. Hardware drivers are required to utilize different hardware features, for example, the invasion of hardware resources or data transfer through direct memory access hardware accelerator. In a similar way, OctoPOS provides an interface to the network interface which is used by the software layers to benefit from the proposed contributions of our work. In chapter 4, the interface details of the OctoPOS with proposed network interface would be presented.

2.4.3. Hardware

The hardware architecture which contains features to support the principles of invasive computing is referred as *InvasIC* architecture. An incarnation of InvasIC architecture is shown in the figure 2.14.

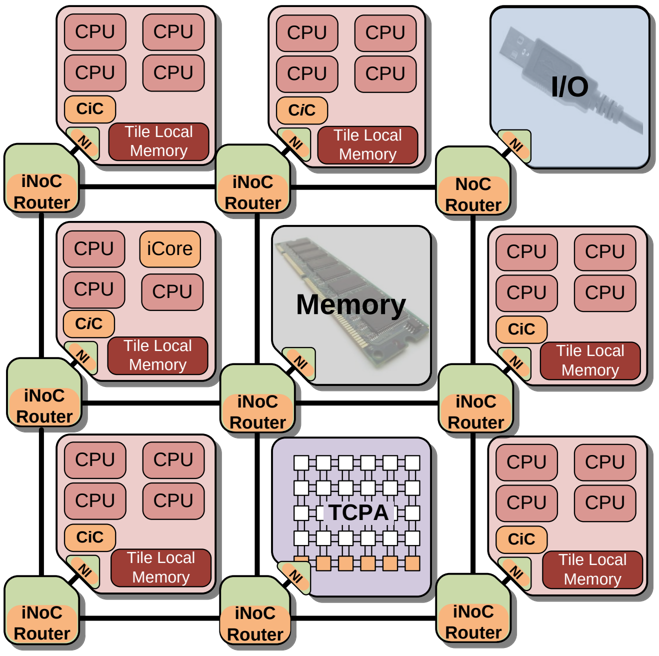


Figure 2.14.: InvasIC architecture consisting of compute, memory and I/O tiles interconnected through Network on Chip

InvasIC is a heterogeneous tiled many-core architecture, which consists of different types of tiles including processing, memory and I/O tiles [63]. The processing tiles contain compute elements including RISC processors, Tightly-Coupled Processor Arrays (TCPAs) and reconfigurable RISC cores (i-Cores). All tiles are connected by the Invasive Network on Chip (iNoC) in a 2D mesh topology. The proposed network interface design provides the interface between the tile interconnect and the iNoC router. The architecture is implemented as a distributed shared memory architecture. Following sections provide a brief overview of different components of the InvasIC architecture.

2.4.3.1. Tiles

The invasive architecture consists of following different types of tiles:

Standard RISC Cores Tiles : The standard RISC core tiles also referred as loosely coupled MPSoC tiles, consist of Leon3 processing cores [43]. Leon3 processors are based on SPARC V8 instruction set architecture. These processing cores were selected to be used in the project due to the less implementation complexity and open source availability [44]. The number of processing cores in a tile is configurable. Figure 2.15 shows the architecture of a standard RISC core tile. Each core contains a separate L1 data and instruction cache. The L1 caches of the processors are connected to the tile bus. The tile bus follows AMBA AHB protocol as detailed in the section 2.1.1. Tile local memory is also attached to the AHB Bus, which contains frequently accessed application code or data. It is an SRAM-based on-chip memory with single cycle access latency.

L2 cache is another component attached to the tile interconnect. It is shared between all cores within the tile and is used for holding the data, which is fetched from remote tiles. This data is accessed transparently by the network interface in case of an L2 cache miss. For this purpose, remote memory access support is provided in the network interface. The details of the remote memory access support will follow in the implementation chapter. Besides remote memory access, the network interface offers an additional interface to the tile bus. This interface is used by the software to configure the network interface for services like remote direct memory access, Quality of Service communication and task spawning synchronization support. Another component in the RISC core tile is the Core i-let Controller (CiC) [117]. Incoming i-lets received from the remote tiles are forwarded directly to the CiC by the network interface. CiC is the hardware module, which is responsible for the assignment of i-lets to the processing core within the tile keeping in view the underlying hardware status and application requirements.

i-Core Tiles : i-Core compute tiles include a special processing core named as an i-core. This core in present in addition to the other standard Leon3 RISC cores. i-Core supports special instructions in addition to the standard ISA of the LEON3 SPARC V8 architecture [62], [43]. These instructions are designed to accelerate the execution of certain applications. Special instructions reconfigure the processor micro-architecture to extract higher performance. i-Core combines the concepts of adaptive micro-architecture and fine-grained reconfiguration to instantiate application-specific accelerators at run-time. The reconfigurable fabric is an integral part of the i-Core functionality. It represents the reconfigurable logic area which is loosely connected to the processor pipeline. The reconfigurable fabric requires a dedicated high bandwidth connection to the tile local memory. Once the reconfiguration is done, the accelerator in the reconfigurable fabric can be accessed through the special instructions. Depending on the application, significant performance improvement can be achieved by using the i-Core special instructions.

Tightly-Coupled Processor Arrays Tiles : Tightly-coupled processing elements structured in a 2D array manner constitute the TCPA tile [81]. Communication between the pro-

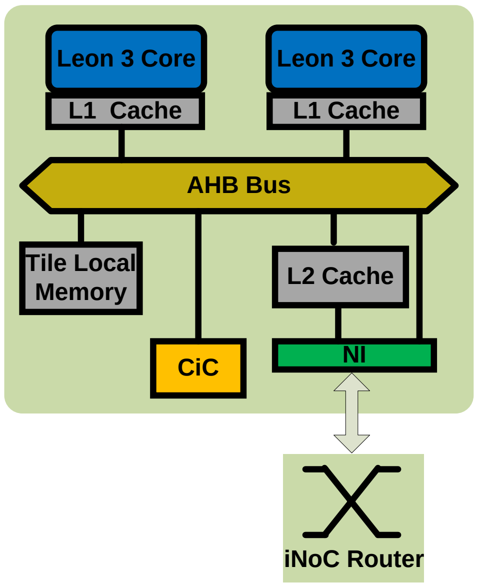


Figure 2.15.: Standard RISC Core Tile in InvasIC architecture

cessing elements is supported via point-to-point connections. TCPA structure is highly suitable for the execution of loop parallel applications like pixel based image correlation. Each processing element represents a Very Long Instruction Word (VLIW) processor and a hardware unit named as invasion controller. The invasion controller enables the hardware-assisted invasion of processing elements to minimize the overhead of invasion. TCPA tile is connected with the iNoC router via the network interface. A LEON3 RISC core is used to handle the communication between the TCPA and the remaining tiles in the architecture. An AMBA AHB bus is used to connect the TCPA, the LEON3 core and the network interface.

Memory and I/O Tiles : Memory tile contains the off-chip memory, whereas the I/O tile provides an interface to the standard I/O peripherals. The off-chip memory of the architecture is composed of multiple memory tiles. Each memory tile contains a DDR controller to interface the external memory over an AHB Bus. The network interface connects the memory tile to the Network on Chip router. A LEON3 core executes an OctoPOS instance for handling communication with other tiles. I/O peripherals include Ethernet, DVI, and custom AHB transactor interfaces. I/O tile enables the interface of such peripherals in a NoC based system. These peripherals are used to transfer data between the external world and the architecture.

2.4.3.2. Invasive Network on Chip

Invasive Network on Chip is the communication infrastructure to connect different types of tiles in the InvasIC architecture [53]. The architecture of an iNoC router is shown in the figure 2.16. The Invasive Network on Chip is a packet switching based Network on Chip,

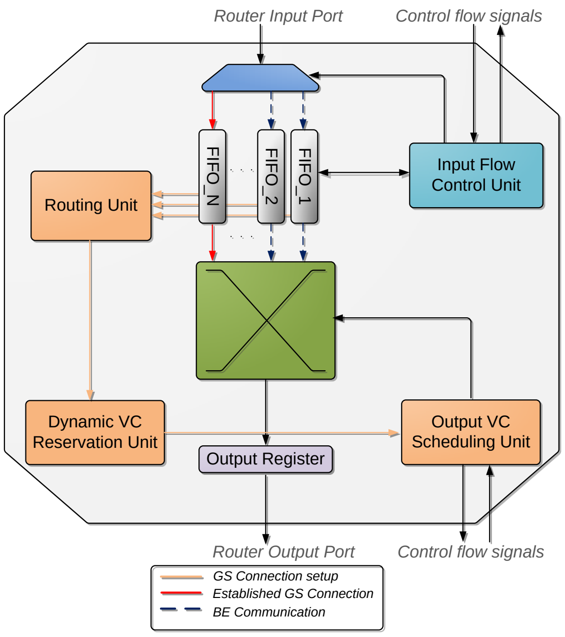


Figure 2.16.: iNoC Router architecture

which also supports virtual channels. It uses wormhole switching to save buffer space inside the routers. For each virtual channel, a FIFO is placed at the input port, which provides the temporary storage for the flits before forwarding. Registers are placed at the output ports to avoid long combinational path. iNoC router uses a meshed topology with bidirectional links and supports XY routing. All aspects of the iNoC are designed with a focus on scalability and distributed resource management. In addition, iNoC advocates the concept of distributed communication resource management to fulfill the application's requirements [59], [58]. State of the art packet switched router architecture is enhanced to enable assignment of communication resources at run-time according to applications demands.

iNoC supports the invasion of communication resources by providing Quality of Service guarantees over the network. QoS is provided by establishing guaranteed service connections over the network. End-to-end virtual channel reservation is performed to establish a GS connection [55]. Weighted round robin scheduling policy as described in section 2.2.5 is deployed to support scheduling of GS and BE traffic over the link. According to this arbitration policy, different traffic flows can be assigned different weights according to their

QoS requirements. iNoC router uses VC reservation for GS communication. In addition to distributed communication resource management and QoS, iNoC addresses important communication aspects for next generation interconnects like resource monitoring and fault tolerance [57], [54].

Distributed shared memory architectures face two important challenges for meaningful exploitation of task level parallelism i.e. communication resource management and synchronization support for software. Concerning communication resource management, conventional approaches disregard the dynamics of underlying platform while assigning virtual channels among applications, which leads to inefficient communication infrastructure utilization and sub-optimal performance. We have addressed these concerns in our approach, which will be detailed in the next chapter. The state of the art methods investigating synchronization support target customized programming models and architectures. The applicability of these approaches for scalable DSM architectures and/or standard shared memory and message passing programming models is not considered. In addition, state of the art concepts do not address task spawning overhead. In this work, our investigations have been carried out under the umbrella of invasive computing paradigm. However, they are applicable to DSM architectures and programming models in general. In addition, we have addressed synchronization support for task spawning, which has not been addressed by state of the art concepts.

3. Communication Resource Management and Software Communication Support

In this chapter, we will discuss our contributions to address the two very important challenges in the distributed shared memory architectures. Those challenges were briefly described in the introduction chapter. The first challenge is efficient communication resource allocation between guaranteed service and best effort communication. Minimizing the synchronization delays during inter-tile data transfer and task spawning operations represents the second issue.

3.1. Communication Resource Management

The proposed communication resource allocation methodology is inspired from the self-optimization principle and depends on the monitoring of real-time traffic data over the underlying DSM platform. Following are the two factors which are the motivation to rely on the monitoring information instead of using prior knowledge of application's communication behavior in our concept.

1. Modern real world applications exhibit heterogeneous communication requirements on the underlying architecture. As an example, we have considered a real world application Video Object Plan Decoder (VOPD) [9]. Tasks and the communication between those tasks in the VOPD application are represented in the form of a so-called application core graph in the figure 3.1. Application core graph represents computation tasks as nodes which can be mapped on independent processing tiles in a many-core system. The communication along with the average rate of data transfer between tasks is shown by the weighted edges in the graph. Figure 3.1 highlights the heterogeneity in communication requirements between different application tasks. This diversity results in different communication bandwidth requirements between the processing resources during the execution of the application. Static knowledge about the application's data flow behavior can not be used to predict the communication bandwidth requirements in a given time interval which is only a segment of the application's lifetime.
2. In DSM architectures, platform resources are shared between concurrent applications. Figure 3.2 illustrates a scenario where three concurrent applications are running on the same underlying architecture and competing for computation and communication resources. Concurrent applications make it difficult to predict the runtime data flow patterns between tiles by relying on the static knowledge of individual application's communication behavior.

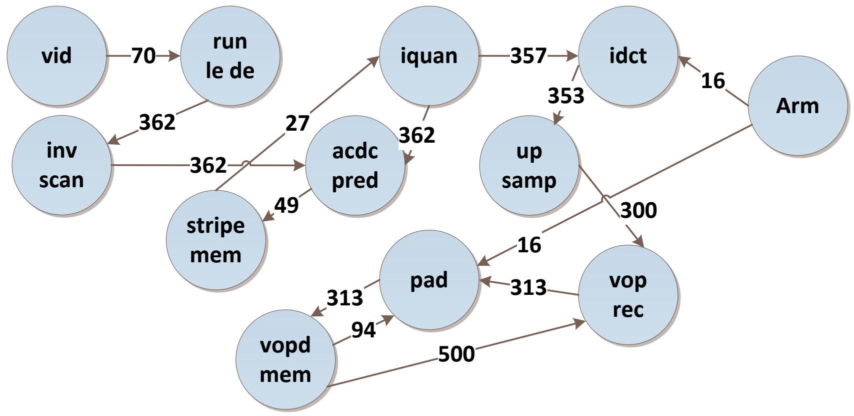


Figure 3.1.: VOPD core graph with varying communication bandwidth requirements [9]

The above-mentioned two factors follow the same argumentation line which emphasizes the presence of caches in the processing architectures. Caches improve the system performance by taking into account the run-time access patterns instead of relying on prior knowledge of application's data access behavior. Keeping in view these aspects, it is evident that the state of the art methods which rely on the static information of application's data flow behavior, lead to the inaccurate characterization of run-time communication behavior. As a consequence, the communication resource allocation performed by such approaches results in the sub-optimal utilization of Network on Chip resources. It leads to the conclusion that efficient communication resource management can not be ensured without considering dynamic traffic behavior in DSM architectures.

Assigning communication resources by monitoring the underlying platform status relates to the principle of self-optimization. However, dynamically changing communication patterns generated by applications pose a challenge for realizing a suitable self-optimization approach. For software-based self-optimization approaches, the overhead associated with dynamic traffic characterization and the subsequent resource management is very high [115], [88]. This leads to the requirement of an hardware-based self-optimization scheme for communication resource allocation. Therefore, the first important constituent of our approach is hardware-controlled self-optimization.

In real world scenarios, it can be observed that most of the applications exhibit temporal locality in terms of their communication behavior. This behavior is observable in the form of end-to-end communication where a source node communicates with a subset of destination nodes. Hence, exploiting the communication locality in data access patterns is the second ingredient of our approach for managing Network on Chip resources. Based on self-optimization and communication locality ideas, we have come up with a concept which optimizes the utilization of communication infrastructure by allocating resources on the basis of run-time traffic information [144]. The details about our approach are given in the following section.

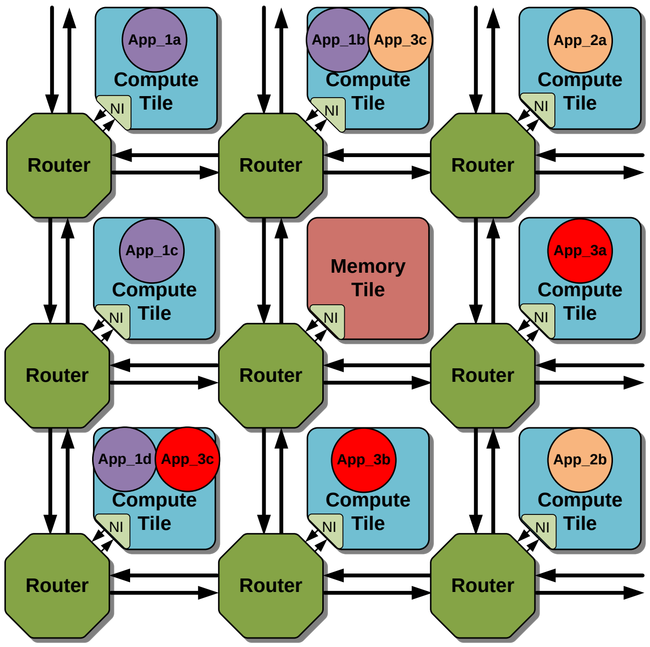


Figure 3.2.: Concurrent applications sharing an underlying DSM architecture

3.1.1. AUTO_GS:Hardware-controlled GS connections

The proposed hardware-controlled self-optimization mechanism to manage communication resources is established over a Network on Chip based DSM architecture. The proposed concept is based on extending the state of the art network interface design in a manner that it can support the setting up of guaranteed service connections autonomously. According to our concept, communication locality between the tiles is detected by monitoring traffic patterns in the network interface of each tile for a configurable time interval. On the basis of this monitoring information, the network interface gains the knowledge about the temporal locality of communication requests which are sent by this tile. Afterward, the network interface exploits this information to establish GS connections to the tiles which are frequently accessed. The less frequently accessed tiles are served via best effort traffic. Best effort communication is also the default communication mechanism in the absence of guaranteed service connections. We have chosen the network interface for our concept because it is the most suitable component to track the communication history of a given tile. In addition, the support provided in the network interface architecture can be configured to adapt itself according to dynamic communication behavior.

Proposed hardware-controlled GS connections lead to better utilization of communication infrastructure which results in reducing the power consumption of the Network on Chip. In addition, they lower the latency for the traffic which is routed towards the frequently accessed nodes and hence minimizing the network congestion. Our concept enables the usage of GS connections between the nodes which show high communication affinity to each other. In this manner, the benefits of low latency communication are made available to the best effort applications. Our methodology also ensures that the hardware-controlled connections do not reserve all communication resources which may lead to complete starvation of best effort traffic in the network. We have referred our approach as *AUTO_GS*. The frequently accessed tiles are called hotspot tiles. The tiles which are accessed less frequently are referred as cold-spots. Our concept to assign GS connections to the hotspots while serving the cold-spots via best effort resources by the network interface is shown in the figure 3.3.

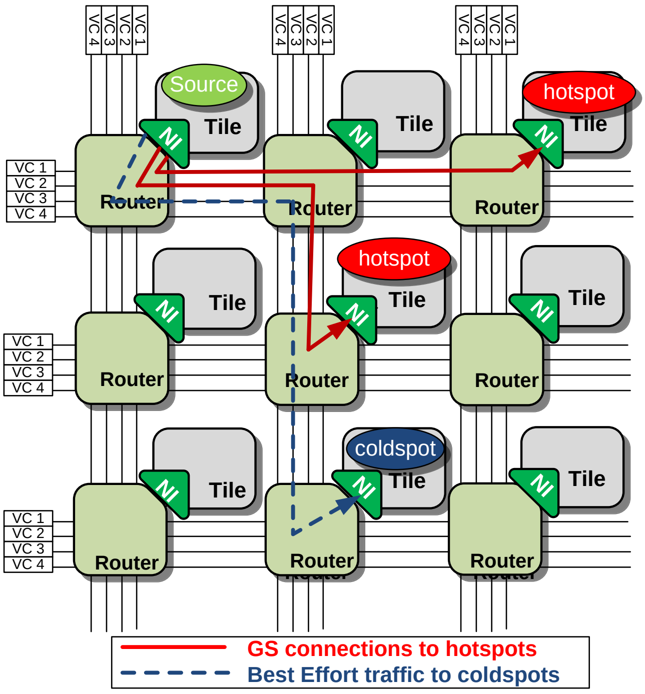


Figure 3.3.: Communication resource management through hardware-controlled GS connections [144]

The proposed self-optimization approach for assigning communication resources consists of three phases: 1) monitoring of communication temporal locality, 2) sorting the target tiles in the order of higher communication locality, and 3) management of GS connections based on the monitoring information. The details about these concept phases are provided in the following sections.

3.1.1.1. Monitoring communication locality

In our concept, the monitoring of communication behavior is done on a packet basis. Through monitoring, a history of communication for all packets which are leaving the network interface is maintained. The history of communication is stored in the network interface for a given time interval $AUTO\_GS_{cycle}$ which is set at the design time. The lower bound on the monitoring interval is given by the following relation 3.1.

$$AUTO\_GS_{cycle} \geq T_{setup,worst\_case} + T_{release,worst\_case} \tag{3.1}$$

The lower bound on the monitoring interval is defined by keeping in view the worst case scenario where an already established connection between the two nodes located at maximum hop distance is released and a new connection for the same hop distance must be established. $T_{setup,worst\_case}$ is the worst case connection setup time between two nodes which are located at the maximum hop distance in the Network on Chip whereas $T_{release,worst\_case}$ refers to connection release time for the same hop distance. $T_{setup,worst\_case}$ is given by the following equation.

$$T_{setup,worst\_case} = T_{setup,worst\_case\_request} + T_{setup,worst\_case\_ack} \tag{3.2}$$

$T_{setup,worst\_case\_request}$ and $T_{setup,worst\_case\_ack}$ describe the worst case connection request and connection acknowledgment delays respectively. Worst case connection request and connection release delays can be further elaborated by the following equation.

$$T_{setup,worst\_case\_request} = T_{release,worst\_case} = hop\_count\_max \times (T_{arbitration,worst\_case} + T_{proc}) \tag{3.3}$$

hop_count_max represents the maximum hop distance over the Network on Chip. T_{proc} specifies the processing delay for forwarding the flit including transmission, output port reservation, and scheduling delays. $T_{arbitration,worst\_case}$ refers to the worst case arbitration delay experienced by a packet in the router. The worst case scenario occurs when a given packet has to wait in the virtual channel buffer because all input ports want to send packets to the same output port. This condition is represented by the following equation.

$$T_{arbitration,worst\_case} = ((N-1) \times VC\_CNT \times pkt\_size) \tag{3.4}$$

Where N and VC_CNT refer to the number of router ports and number of virtual channels per port respectively. pkt_size denotes the number of flits in a packet. Successful establishment of a GS connection requires an acknowledgment from the destination tile. Acknowledgment for the connection setup consists of a single flit which is sent over a separate physical network. This network is reserved for the connection acknowledgment traffic. Worst case connection acknowledgment delay $T_{setup,worst\_case\_ack}$ is given by the following equation.

$$T_{setup,worst\_case\_ack} = hop\_count\_max \times ((N-1) + T_{proc}) \tag{3.5}$$

Connection release is a non-blocking process i.e. no acknowledgment is required. The implementation details of connection establishment and release processes are provided in section 4.2. $AUTO\_GS_{cycle}$ is set to a value which is the integer multiple of the sum of $T_{setup,worst\_case}$ and $T_{release,worst\_case}$ delays as well as the subinterval of the application's execution time. Average flit injection rate of the application is another parameter which influences the selection of the monitoring interval. The impact of the monitoring interval selection on the performance of our concept is quantitatively evaluated in the section 3.1.2.

After the time $AUTO\_GS_{cycle}$ is passed, the communication history for the tiles is analyzed in the next phase. The previous communication history is cleared and the behavior is sampled till the next time interval. The number of target tiles for which the communication history is maintained can be defined by the following relation.

$$Tiles_{total} \geq AUTO\_GS_{conn} \tag{3.6}$$

Total number of tiles in the architecture is given by the parameter $Tiles_{total}$. $AUTO\_GS_{conn}$ gives the number of tiles with whom the given tile may possibly communicate. This parameter represents the number of communication partners of the considered tile. Application core graph, which was shown in the figure 3.1, is used to determine the number of communication partners. The number of monitored communication partners is the second important parameters of our concept. The quantitative evaluation of the parameter $AUTO\_GS_{conn}$ is also discussed in the section 3.1.2.

3.1.1.2. Analyzing communication history

In this phase, the communication history obtained during the first phase is analyzed and the target tiles are sorted in the order of increasing communication locality. In addition, it is checked if the individual entries qualify for being served via connection-oriented traffic. A destination tile qualifies for an AUTO_GS connection if it receives the amount of data which is above the given threshold of the communication bandwidth of the source tile. The threshold for that data amount is represented by the number of packets per measurement cycle time $AUTO\_GS_{cycle}$ and is represented by the relation 3.7.

$$Tile_{tgt\_curr\_data} \geq Link\_BW / AUTO\_GS_{conn} \times pkt\_size \tag{3.7}$$

Where $Tile_{tgt\_curr\_data}$ represents the data sent to the destination tile which is being evaluated as a candidate to be served via GS connection. $Link\_BW$ gives the total number of flits sent out by the source tile in the monitoring interval. If a target tile satisfies the condition represented by the relation 3.7, it is considered as a hotspot. The tiles which do not qualify against this condition are marked as cold-spots.

3.1.1.3. Establishing AUTO_GS connections

In this phase, the GS connections are set up after the communication history analysis phase is over. Connections are established to the tiles which were marked as hotspots. Only those hotspot tiles are considered for new connections for which the GS connections do not exist. In order to prevent complete starvation of best effort traffic, one virtual channel is always left over for the BE communication. Equation 3.8 describes this condition.

$$AUTO\_GS_{max} = VC\_CNT - 1 \tag{3.8}$$

$AUTO\_GS_{max}$ denotes the maximum number of AUTO_GS connections which can be established. If there are more hotspots than the maximum allowed connections, the utilization of existing AUTO_GS connections is evaluated according to the relation 3.7. In the case where the established GS connection is found to be underutilized, it is released. If there are more hotspot candidates than the $AUTO\_GS_{max}$ even after releasing the connections to the cold-spots, the ones with higher data share $Tile_{tgt\_curr\_data}$ are served via connection-oriented traffic. Once the GS connections are established for the hotspots, they are allocated a fixed share of the link bandwidth. The remaining candidates are still served via best effort traffic. The details specific to the implementation of our concept are provided in section 4.2.

3.1.2. Evaluation using SystemC Modeling and Simulation

Cycle accurate parameterizable SystemC models of the network interface and NoC router are implemented to evaluate the proposed AUTO_GS concept. SystemC-based framework results in lesser modeling effort and faster simulation time as compared to the RTL simulation. For AUTO_GS concept, the Network on Chip related parameters are set to a fixed configuration. The purpose of this was to focus on the evaluation of AUTO_GS parameters. A 10x10 mesh based Network on Chip is used for the investigation of the AUTO_GS concept. Network on Chip router model represents the functionality of iNoC as described in section 2.4.3.2. XY routing and wormhole switching are set as routing and switching techniques respectively. Link transmission delay of 2 cycles, VC reservation delay of 1 cycle and throughput of 1 flit/cycle/port is assumed. These numbers are chosen to get results comparable to a hardware implementation. The number of virtual channels in the network interface and NoC routers is set to 4. The VC buffer depth is configured to hold 4 flits. In addition, the packet size of 6 flits is chosen. Best effort communication is set a default for all virtual channels unless the GS communication is requested.

For each scenario, the results are obtained by taking an average of ten independent simulations over 10^6 cycles. Network on Chip clock frequency is set to 100MHz. Measurements are started after a warm up time of 10,000 cycles. A protocol overhead of 1 flit per packet is assumed for packet switching communication as compared to the connection-oriented communication. This is a valid assumption, since additional information such as source and destination network address contained in the head flit, need to be included in each packet for best effort traffic. Compared to the best effort traffic, GS connections have no protocol overhead once the connection is established. Two configurations are compared against each other: As a reference, a software-controlled communication resource management approach is considered [20]. It is named *Reference* in the evaluations. The above-mentioned approach analyzes the run-time traffic information of the tiles in a centralized software instance. The nature of communication between different nodes, guaranteed service or best effort, is chosen according to the decisions of the software instance.

For the second configuration, the proposed hardware support in the network interface is considered. The configuration related to our concept is referred as *AUTO_GS*.

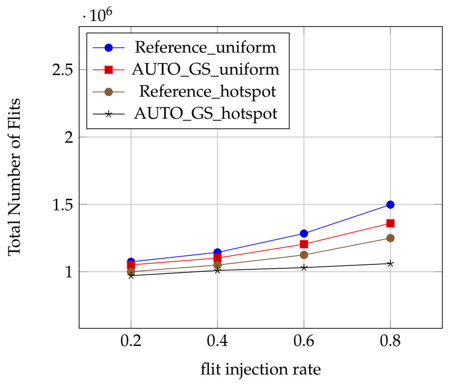
3.1.2.1. Synthetic Traffic

Synthetic traffic is used as input stimuli for the following investigations. We have used two different traffic patterns, which are widely used in Network on Chip related evaluations [100] [106] [45]. The first type of traffic pattern is known as uniform random in which each node sends data to other nodes in the architecture with same data rate. However, the selection of the nodes in the architecture is done on a random basis. The other traffic pattern is the hotspot, which belongs to the class of non-uniform traffic patterns and exploits the communication locality between network nodes. In hotspot traffic pattern, each node sends a major percentage of its traffic to a given set of node(s). Whereas, the remaining percentage of the traffic is uniform random. The data rate is controlled by the parameter called flit injection rate. Flit injection rate is defined in flits/cycle/node. It can be varied per node between 0 and 1 flit per cycle.

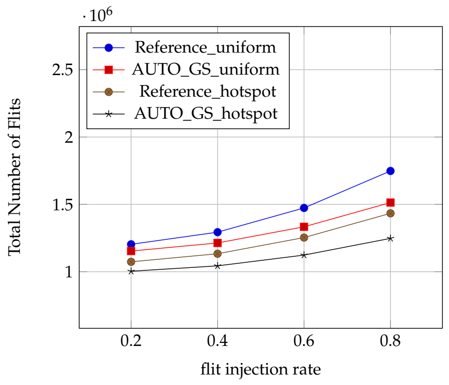
For synthetic traffic based evaluations, the number of communication partners per node $AUTO\_GS_{conn}$ is varied. Keeping in view the above-mentioned Network on Chip parameters for simulation, $AUTO\_GS_{cycle}$ is set to 4160 clock cycles according to the relation 3.1. It comes out to be 41.6μsec regarding the clock frequency of 100MHz. Keeping in view the number of virtual channels, $AUTO\_GS_{max}$ is fixed to 3 for following investigations. One virtual channel is always reserved for best effort traffic.

At the start, when no GS connections exist, best effort is the default communication mechanism. The software-based Reference configuration and the proposed hardware-based AUTO_GS concept monitor the frequently used communication flows and establish GS connections to them. In three simulation scenarios, the number of communication partners i.e. $AUTO\_GS_{conn}$ is varied. Within each scenario, the flit injection rate is increased linearly. We have compared the utilization of Network on Chip in both Reference and AUTO_GS configurations. The utilization is measured in terms of the number of flits which are transmitted. Figure 3.4 shows the total number of flits transmitted by the Reference and AUTO_GS mechanisms for both uniform and hotspot traffic types. Reference_uniform and Reference_hotspot denote the results of the Reference configuration for the two applied traffic types. AUTO_GS_uniform and AUTO_GS_hotspot represent the values of AUTO_GS configuration for uniform and hotspot traffic patterns respectively.

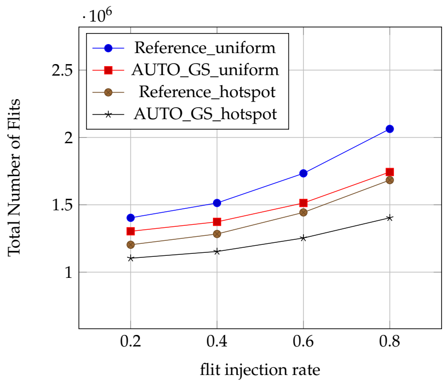
AUTO_GS configurations perform better than the corresponding Reference configurations for both uniform and hotspot traffic. The difference between Reference and the proposed concept comes from the fact that the AUTO_GS configuration utilizes low protocol overhead GS communication in more efficient manner. AUTO_GS hardware support requires smaller time to make decisions for communication resource management and thus routes more traffic via connection-oriented communication. For a given number of communication partners, the number of flits increase with the increase in injection rate for all configurations. The increase in the transmitted data at higher injection rates is more observable for uniform traffic as compared to the hotspot traffic. This is due to the reason that the traffic is uniformly distributed in the uniform traffic pattern and the number of virtual channels which can be used for connection-oriented traffic are limited. When the number of communication partners are increased from 4 to 8, the amount of traffic in increased to 38% for Reference_uniform. As compared to AUTO_GS_uniform, AUTO_GS_hotspot reduces the NoC utilization to a greater extent especially when the number of communication partners increase. This happens due to the fact that hotspot traffic leads to skewed communication patterns in which some nodes receive a larger share of traffic. These nodes become ideal candidates for being served via the AUTO_GS connections and hence result in getting more benefit from AUTO_GS concept. When $AUTO\_GS_{conn}$ is set to 8, the NoC utilization can be reduced up to 20% in AUTO_GS_hotspot configuration compared to the Reference_hotspot.



(a) $AUTO\_GS_{conn} = 4$



(b) $AUTO\_GS_{conn} = 6$



(c) $AUTO\_GS_{conn} = 8$

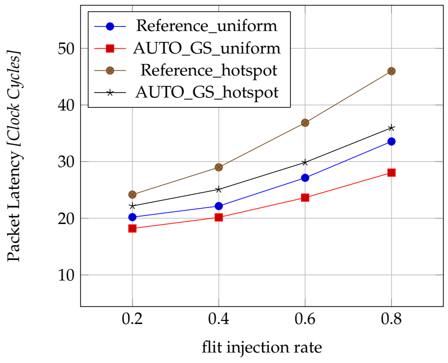
Figure 3.4.: Network utilization for synthetic traffic

Figure 3.5 shows the average packet latency analysis for the described evaluation setup. $AUTO\_GS_{conn}$ is varied between three different values from 4 to 8. Within each scenario, flit injection rate is scaled linearly from 0.2 to 0.8. For a given number of communication partners, higher packet latency is observed with the increase in injection rate as the network starts getting saturated. In general, both Reference configurations show higher average packet latency compared to the AUTO_GS configurations. This is due to the fact that the Reference configurations require more time in order to react to the run-time traffic changes as compared to the AUTO_GS hardware support. As a consequence, Reference configurations route larger share of traffic through best effort communication instead of using GS connections. The high share of best effort communication results in the increased hop latency which is due to the additional reservation delay of 1 cycle per node for each packet. Whenever the best effort head flit enters into a router, the buffer inside the router has to be temporarily reserved for this packet. In the case of GS communication, the reservation is required only at the start of the connection. Therefore, the subsequent flits do not experience the reservation latency. Both Reference and AUTO_GS configurations show lesser packet latency for uniform traffic as compared to the hotspot traffic because the hotspot traffic leads to higher network congestion. For injection rate of 0.8, AUTO_GS shows latency reduction of up to 28% as compared to the Reference for hotspot traffic. When the number of communication partners is increased from a lower value to a higher value, average packet latency reduces for a given injection rate. This is due to the reason that the outgoing traffic gets distributed between a relatively higher number of nodes.

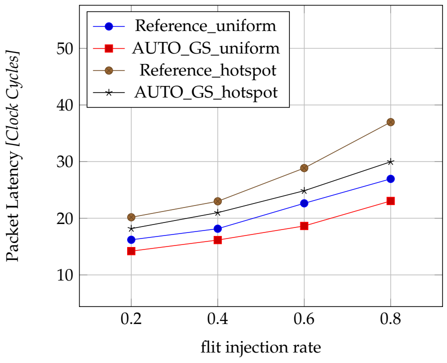
Finally, the energy consumption for data transmission is analyzed for the Reference and AUTO_GS configurations. For this purpose, the term Communication Related Energy (CRE) consumption is introduced. CRE is defined as the energy consumption, which is directly related to the data transmission in the network. The energy consumed by the Network on Chip router and the network interface is measured by post-synthesis power analysis. Value change dump files are used to get the toggle rates. Afterward, *Synopsys Power Compiler* is used to estimate the power.

The simulation parameters, which were used for the network utilization and latency evaluations, are also used for the energy consumption measurements. The number of monitored communication flows per node are varied from 4 to 8. The most frequently used flows are detected and GS connections are set up for them. However, the number of connections which can be established is limited to 3 according to the equation 3.8. In our case, the idle energy consumption for the NoC router was measured to be 7.5mW. Whereas the network interface consumes the idle energy of 6.1mW. Idle energy consumption consists of the static power as well as dynamic power resulting from the clock signal. The difference between the total and the idle energy consumption is the communication-related energy consumption, which is plotted in figure 3.6. The communication-related energy has been focused in these results as this is the energy consumption which is influenced by our concept. Static energy consumption is agnostic from the concept and hence has not been analyzed.

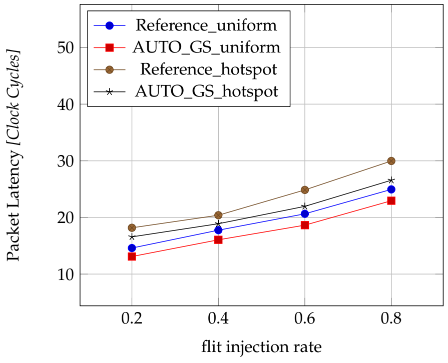
AUTO_GS configuration saves more interconnect power compared to the Reference. In the case of injection rate of 0.8, AUTO_GS can reduce the CRE by 26% compared to the Reference configuration for hotspot traffic. The reduction in the saved energy relies on the fact that the amount of best effort communication is higher in the Reference configuration as compared to the AUTO_GS. Best effort communication consumes more energy because of the reservation process which has to be done for every best effort packet in each subsequent router in the path. In addition, the best effort protocol overhead leads to



(a) $AUTO\_GS_{conn} = 4$



(b) $AUTO\_GS_{conn} = 6$



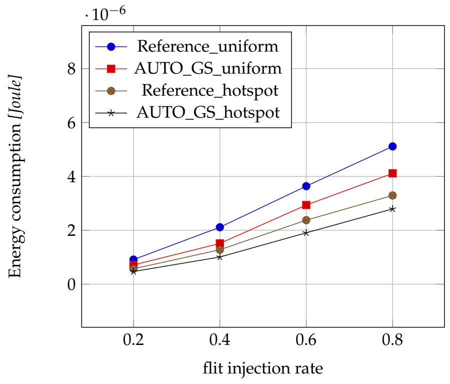
(c) $AUTO\_GS_{conn} = 8$

Figure 3.5.: Average packet latency for synthetic traffic

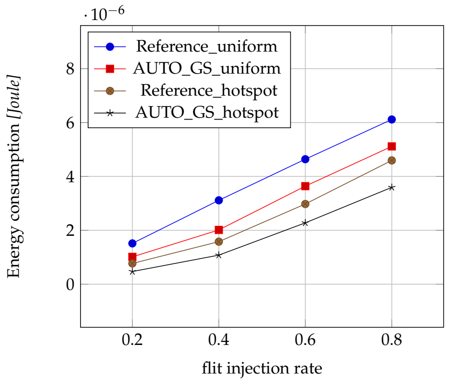
the transmission of additional flits which results in higher energy consumption as compared to GS traffic. With the increase in injection rate, the difference between the two approaches becomes more observable as shown in the figure 3.6. AUTO_GS_hotspot shows more energy saving as compared to AUTO_GS_uniform because of the higher number of flits using the connection-based traffic.

3.1.2.2. Real World Applications

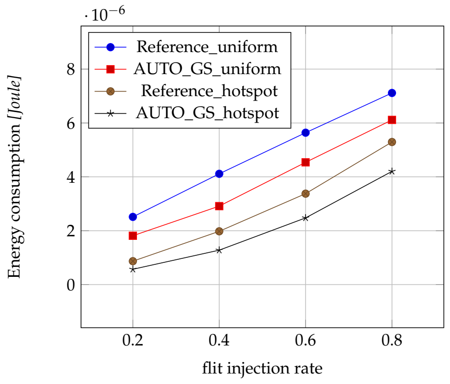
Four real world applications are selected for the investigation of AUTO_GS concept: Video Object Plan Decoding (VOPD), MPEG4 video decoding, Picture-In-Picture (PIP) and Multi-Windows Display (MWD). These high-end video processing applications possess diverse communication requirements and are widely used for System on Chip performance evaluations [133] [98]. Bertozzi et al. represented the communication behavior of these appli-



(a) $AUTO\_GS_{conn} = 4$



(b) $AUTO\_GS_{conn} = 6$



(c) $AUTO\_GS_{conn} = 8$

Figure 3.6.: Communication related energy consumption for synthetic traffic

cations in the form of application core graphs [9]. They have also proposed the mapping of the application tasks on the processing cores through an algorithm, which optimizes the communication between the cores. We have used the same application task mapping in our simulation framework which is proposed by the above-mentioned state of the art concept. The application task mapping results in the allocation of a given number of cores to each application. Mapping of an application to the number of processing cores and the application execution time while using only best effort communication are provided in the table 3.1. Each application is mapped independently to the center of the mesh for each simulation run. The number of monitored communication flows $AUTO\_GS_{conn}$ is chosen by keeping in view the assigned processing cores to each application. $AUTO\_GS_{cycle}$ is changed to three different values in respective different scenarios to check its impact on the performance of our concept. In the first scenario, $AUTO\_GS_{cycle}$ is set to 41.6μsec, which represents the lowest possible value of monitoring interval according to the rela-

Application	Number of cores	Execution time (ms)
VOPD	12	5.6
MPEG	14	6.3
PIP	8	4.3
MWD	14	5.7

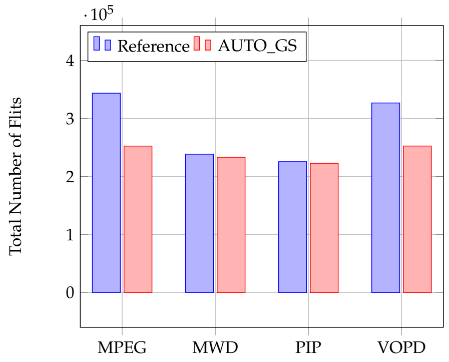
Table 3.1.: Real world video processing applications

tion 3.1. Subsequently, the monitoring interval is increased to multiple of this value in two further scenarios. Monitoring interval is always less than the execution time of each application. The remaining simulation parameters are chosen to be the same as in the case of synthetic traffic evaluations.

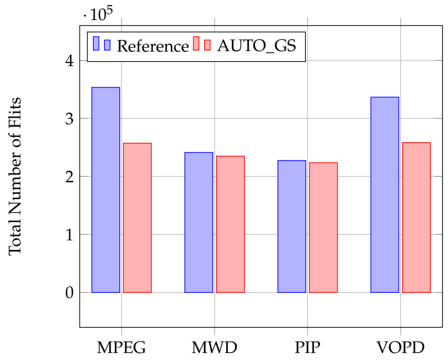
Figure 3.7 shows the total amount of traffic which is generated by the four applications over the network. AUTO_GS configuration leads to significantly less traffic for real world applications when compared with the Reference configuration. In the Reference configuration, a centralized software instance decides to use GS or BE communication depending on the monitoring information. In the proposed AUTO_GS configuration, the hardware support in the network interface is responsible for assigning the communication resources. Hardware-controlled GS connections react faster to the run-time traffic conditions as compared to the Reference and hence major share of the traffic uses connection-oriented communication. The difference in the amount of traffic reduction between the two configurations is different for each application, which depends on its communication behavior. MPEG and VOPD applications have higher communication demands and possess heterogeneous communication requirements between different nodes in comparison to other two applications. Therefore, both of these applications show more reduction in network utilization compared to PIP and MWD, when AUTO_GS concept is applied. When the monitoring interval is increased in three different scenarios, the saving in the amount of traffic is reduced for MPEG application. Other applications do not show an observable difference when monitoring interval is changed. This is due to the fact that the MPEG application shows more variation in its communication behavior over time as compared to other applications.

Figure 3.8 shows the average packet latency while executing the four applications. All applications profit from AUTO_GS concept in terms of packet latency. The packet latency of MPEG in the case of the Reference is higher as compared to the other applications. This is due to the fact that the MPEG generates higher network load. AUTO_GS brings a reduction of 35% in packet latency for MPEG application. In the case of VOPD application, the average packet latency is reduced by 26% in AUTO_GS configuration as compared to the Reference configuration. MWD and PIP being low bandwidth applications profit relatively less from our concept as compared to MPEG and VOPD. The impact of increasing monitoring interval can be observed in the form of increased average packet latency for MPEG application. Similar to network utilization evaluations, other applications do not see the observable difference in packet latency when the monitoring interval is changed.

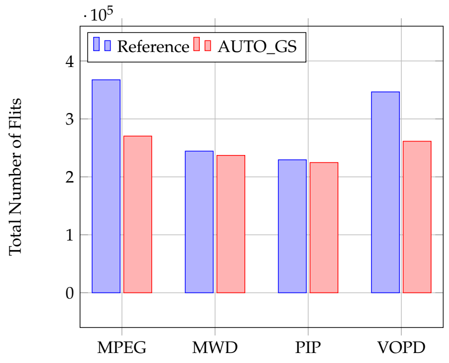
Figure 3.9 shows the energy consumption for the applications. The communication-related energy consumption was analyzed for each application as explained in the evaluations for synthetic traffic. The PIP and MWD applications, which have relatively low bandwidth



(a) $AUTO\_GS_{cycle}$ = 41.6μsec



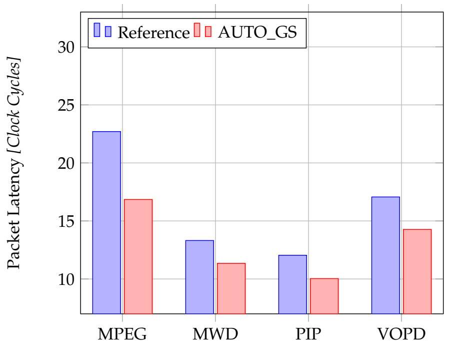
(b) $AUTO\_GS_{cycle}$ = 83.2μsec



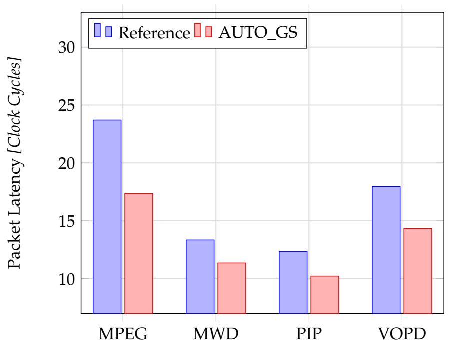
(c) $AUTO\_GS_{cycle}$ = 124.8μsec

Figure 3.7.: Network utilization for real world applications

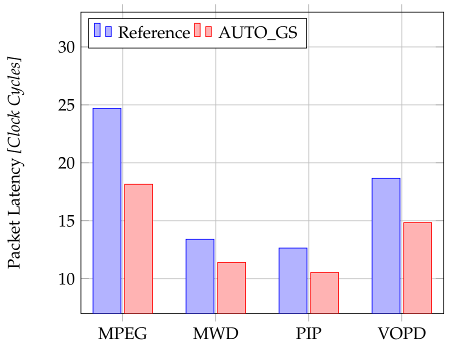
requirements, reduce their communication-related energy consumption by around 25% if AUTO_GS configuration is deployed. VOPD and MPEG show a reduction of around 30% and 33% in consumed energy as compared to the Reference configuration. The amount of energy saving reduces with the increase in monitoring interval for MPEG application following the same reason as stated in network utilization and packet latency results. The evaluations which are presented above, highlight that hardware managed GS connections lead to better utilization of communication resources and reduce the communication latencies suffered by applications. In addition, energy consumed by the communication infrastructure is reduced. The details of the hardware extensions in the network interface design which correspond to the proposed concept are provided in section 4.2.



(a) $AUTO_GS_{cycle} = 41.6\mu$sec



(b) $AUTO_GS_{cycle} = 83.2\mu$sec

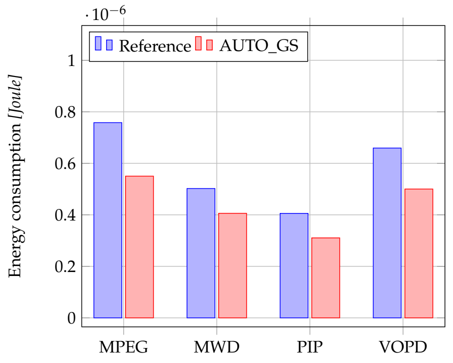


(c) $AUTO_GS_{cycle} = 124.8\mu$sec

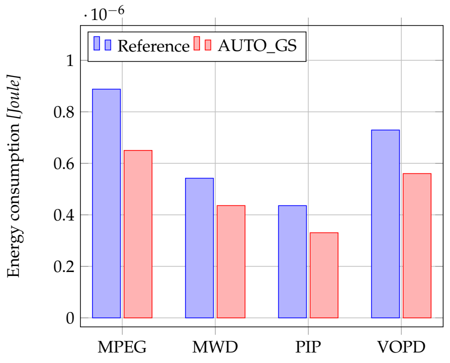
Figure 3.8.: Average packet latency for real world applications

3.2. Inter-tile Software Synchronization Support

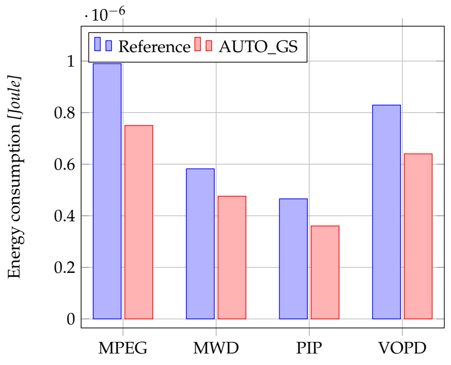
Modern real-world applications benefit from task level parallelism. Through task level parallelism, applications are able to distribute their workload on underlying processing resources. However, the amount of computational workload in such applications is not the same during entire execution time. There is a phase where the application goes through the sequential execution on a single processing resource. Sequential execution represents the state of the application which does not benefit from the distribution of workload on processing resources as stated by Amdahl's law [65]. However, sequential execution is followed by the parallel execution, which desires fine-grained parallelism and hence more computation power is required at this stage. In parallel execution, better performance can be achieved through task level parallelism i.e. by spawning tasks on the available processing resources. The behavior of real world applications with sequential and parallel execution is shown in the figure 3.10.



(a) $AUTO\_GS_{cycle} = 41.6\mu\text{sec}$



(b) $AUTO\_GS_{cycle} = 83.2\mu\text{sec}$



(c) $AUTO\_GS_{cycle} = 124.8\mu\text{sec}$

Figure 3.9.: Communication related energy consumption for real world applications

In order to exploit the available parallelism on a many-core DSM architecture, an application may spawn multiple tasks/processes which can be mapped on the available cores either within the same compute tile or in different compute tiles. The processes mapped within the same compute tile communicate over the shared system bus whereas the processes which are distributed in different tiles communicate over Network on Chip. When tasks belonging to the same application are spawned on the processing nodes which belong to different tiles, they suffer from communication delays. The communication overhead between remote tasks plays a significant role in defining overall application performance. One part of the communication overhead comes from the fact that the remote tasks need to exchange data between them. Another important part of the overhead is associated with distributing and executing the computation workload over the distributed processing resources. We named the operation of distributing computation workload over different tiles as remote task spawning.

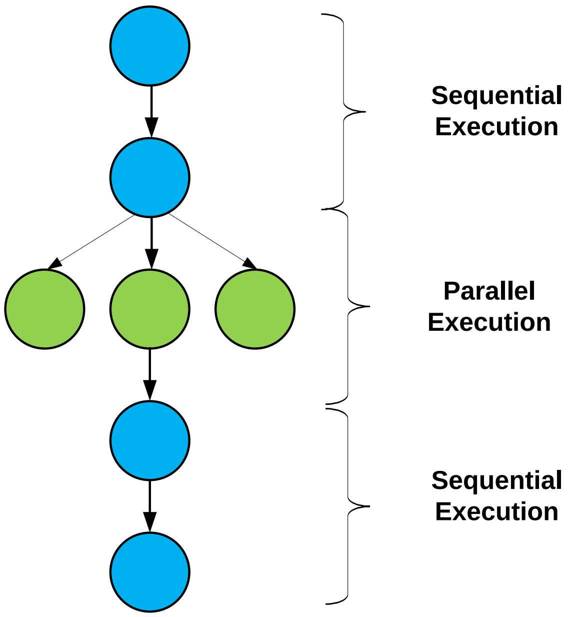


Figure 3.10.: Task level parallelism in real world applications

In a DSM architecture, software layers include distributed operating system and run-time resource management system as introduced in the section 2.4.2. These software layers are required to manage the execution of applications over the distributed platform resources. However, in order to perform the above-mentioned job in an efficient manner, the software needs appropriate support from the underlying architecture. Communication support to facilitate the execution of applications which are mapped on different tiles is the basic requirement towards hardware architecture. Communication support can further be classified into data communication support and task synchronization support for applications. Data communication support includes remote memory access mechanisms as detailed in section 2.3.2. Task synchronization mechanisms are required during workload distribution and execution. In the scope of this work, we have focused on synchronization needed for remote task spawning operation. For data communication and inter-tile synchronization during task spawning, conventional software dominated mechanisms incur large performance overhead. Therefore, in order to exchange data or to achieve synchronization between communicating processes mapped on different tiles, efficient hardware supported mechanisms are desired which offload the software from these duties.

As this work has been carried out in the scope of invasive computing, the software layers are considered, which are described in section 2.4.2. However, the scope of our contributions is not limited to invasive computing and can be applied to any distributed shared memory based platform. In invasive computing, the communication requirements of an application are expressed by the so-called constraints. These constraints are evaluated by the operating system at run-time, taking in consideration the current utilization of the hardware resources including communication infrastructure. The operating system contains low-level driver like sub-routines to configure the communication resources according to the application requirements. Network on Chip including routers and network interface have to support these communication demands accordingly.

Hardware support for remote load/store accesses is required for legacy shared memory programming model. In addition to remote load/stores, modern System on Chip architectures are equipped with remote direct memory access accelerators to increase the transfer efficiency of data copy operations. The above-mentioned mechanisms are supported in most of the state of the art DSM architectures. These mechanisms for data communication are realized targeting high performance for low abstraction communication transactions on the underlying architecture. Therefore, direct utilization of these strategies by the application software results in the high development effort as well as large performance overhead [38]. Keeping in view these aspects, we have designed communication support for software layers in our work. In the scope of assisting software layers over the DSM architectures, the first contribution of this work is the efficient asynchronous data transfer hardware support which can perform programmable data transfer operations over Network on Chip. In addition, the above-mentioned hardware support is capable of handling data transfer signaling internally without requiring software involvement. In order to investigate the data transfer hardware support, we have used the proposed methodology for data prefetching in invasive computing [60]. Details about the data prefetching concept and the associated evaluations will follow in section 3.2.1. The synchronization support for software layers in DSM architectures during task spawning is the second contribution offered by this work [145]. We have introduced remote task spawning hardware support to offload the software during workload distribution. Details of the proposed contribution for remote task spawning will follow in section 3.2.2.

3.2.1. Data prefetching through asynchronous data transfer support

Data access delays make a significant share of overall execution time over Network on Chip based systems. This is due to the fact that the application has to suffer from large communication and synchronization delays while transferring data between remote tiles. Therefore, in order to extract maximum performance from a distributed shared memory architecture, the application developer has to exploit data locality by avoiding frequent NoC accesses. Caches exploit data locality and hence result in improving the system performance. However, because of the limitation of available on-chip memory, caches are helpful for smaller data sets. In addition, cache coherence becomes a major challenge over a distributed interconnect like Network on Chip when the data is shared between multiple tiles and needs to be frequently modified [113]. The above-mentioned considerations highlight the importance of mechanisms which ensure the placement of frequently accessed large data sets close to the processor by supporting low-overhead data transfer operations between tiles.

DSM architectures contain tile local memory in each compute node as shown in the figure 2.15. This memory can be accessed with significantly lesser latency as compared to the remote memory. However, the limited size of this memory because of being on-chip SRAM, emphasizes the need for its efficient utilization. Researchers have proposed to delegate the management of this data storage to the application developer [72]. Application controlled memory management relieves operating system from memory handling operations and hence results in extracting better performance from the local storage. In this manner, the frequently accessed data can be placed in the local memory and the costly data transfers over Network on Chip can be avoided. However, this kind of memory management requires changes in both software and hardware levels. On the software side, the tile local memory handling at the application level should be allowed by the operating system. On the hardware front, the architecture should support direct memory access transfers between remote memory and the tile local memory with minimum software involvement.

In Invasive Computing, an efficient prefetching methodology is conceived to manage data in tile local memory. For real world applications like matrix multiplication or video feature detection, which operate on large sets of data, it is not possible to make the entire data set available in the tile local memory. Therefore, in order to reduce data access delays, the required data sets should be prefetched parallel to the application execution. The prefetching ensures the availability of frequently accessed data in the tile local memory and hence reduces the number of memory accesses outside the tile boundary. As a part of the prefetching support at the software level, the invasive computing proposes changes in the development language. X10 language, which is incorporated in invasive computing, contains explicit *alloc* and *free* function calls. These functions enable the application to have control of the memory management. In order to initiate a data transfer operation to the tile local memory, *LocalMemory* constraint is introduced in X10. Figure 3.11 shows an example X10 code, which shows the data transfer to tile local memory using the prefetching mechanism. In this example, the application developer uses the *future* construct to initiate the data transfer as a prefetching operation. The `force` primitive waits for the data transfer to finish and is used for the synchronization within the application. Keeping

```
val src = TileLocalMemory.alloc[int](cs);
val address = id.ordinal * cs;
val future = data.fetch(address, src);
... // continue with the processing while the data is being copied
    into tile local memory
val dest = future.force();
assert src == dest;
... // use the tile local data in ‘src‘
```

Figure 3.11.: Example code showing the data transfer through prefetching in X10

in view the above-mentioned prefetching methodology, it becomes essential that the efficient mechanisms for asynchronous data transfer between tiles are supported. In state of the art systems, direct memory access is incorporated to move data between memories. The DMA support improves the performance by offloading the software from the data transfer operation. With the introduction of Network on Chip, DMA accelerators are developed which transfer data between different remote nodes [134], [114], [16]. However, the above-mentioned approaches require large software contribution to program DMA accelerator and supervise data transfer handshake. In the above-mentioned research contributions, the subroutines to program the DMA accelerators are heavy operating system calls, which incur large overhead resulting from memory management and data transfer handshake. Therefore, the overhead to utilize the DMA support is significant and may result in performance degradation [4].

In order to support asynchronous data transfer, our approach requires operating system software and hardware changes. As stated above, operating system delegates the management of tile local memory to applications [104]. Therefore, the memory management overhead for data transfer between different tiles through DMA is minimal. In addition, the duty of supervising handshake between different tiles for direct memory access operation is handed over to hardware. Describing further details of the software contributions is not in the scope of this work. On the hardware side, we have introduced an intelligent support on the top of state of the art DMA accelerator which supervises the handshake for data transfer operation without operating system involvement. The DMA support has been positioned inside the network interface, which makes this service uniformly available to all cores in the tile. Application software on the sender side configures the DMA unit via memory-mapped registers. Therefore, the direct memory access support is easily accessible and the overhead to initiate a DMA operation is very less. The introduced hardware support in the network interface performs the bookkeeping of DMA operation internally. This means that the software is completely offloaded from supervising the status of direct memory access on both transmit and receive sides. Direct memory access support in the network interface is capable of transferring data through both best effort and guaranteed service communication. If the Network on Chip is supporting Quality of Service communication for the application, the DMA unit transparently uses the reserved virtual channels for transferring data and hence satisfies the QoS requirements.

3.2.2. Hardware-assisted remote task spawning

We have considered the task spawning operation as a case study to demonstrate the need for efficient synchronization methodologies between remote tiles in DSM architectures. In order to highlight the overheads associated with the existing remote task spawning concepts, we have considered an example. The example presents the state of the art software controlled task spawning operation between two tiles through a message sequence chart. This message sequence chart is shown in the figure 3.12. In the considered example, the task spawning model is assumed, which requires copying of complete code and data associated with the spawned task from source to the destination tile. As represented in the figure 3.12, the remote task spawning could be divided into three sub-operations/steps. In the first step, a software instance on the source tile initiates the remote direct memory access operation to move the task data. When the data transfer is completed, the task pointer which points to the start address of the code in the destination tile is sent by the source tile. Afterward, the software clears the memory on the source tile which was allocated for the spawned task. In such conventional task spawning methodologies, software triggers each subsequent sub-operation after the completion of its predecessor step. Sending of task data and pointer are the operations which require network access

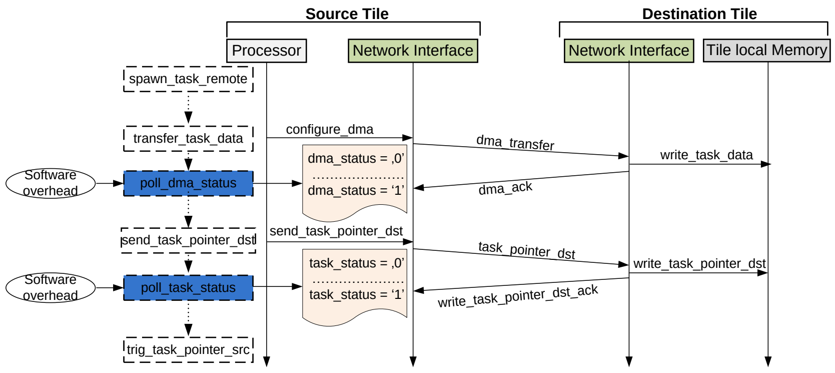


Figure 3.12.: Message sequence chart showing software dominated inter-tile task spawning

and hence suffer from large communication delays. These communication delays result in performance overhead because the software is waiting for the completion of task spawning sub-operations and can not proceed with actual application processing. The impact of task and data pointer exchange delays on the system performance is quantitatively evaluated in the section 3.2.3. Keeping in view above-mentioned aspects, efficient means are required to handle task spawning operations in Network on Chip based architectures.

In Network on Chip based DSM architectures, network interface links computation elements (tiles) and the communication infrastructure (NoC). Therefore, keeping in view its functional significance, we propose hardware support for inter-tile task spawning inside the network interface. According to our concept, the network interface provides the required communication and synchronization support during task spawning. The network interface performs the synchronization related to remote task spawning by handling different phases in hardware and thus offloads the software from synchronization duties. In addition, the presented hardware support handles the synchronization by keeping in view the distributed nature of NoC and thus relieves the tile interconnect from status polling requests. Inter-tile task spawning supported by proposed methodology is presented in the figure 3.13.

In our approach, software is only required to initiate the task spawning request by configuring task spawning hardware support in the network interface. Afterward, the proposed support initiates the transfer of task data by configuring remote direct memory access. Upon successful completion of the DMA operation, the task pointer is sent to the destination tile. When task pointer is sent to the destination, the software on the source side is informed through an interrupt to indicate the completion of task spawning request. The proposed approach delivers higher performance when compared with conventional software-driven approaches because of the fact that hardware holds a larger share of synchronization duties. In addition, only an extension of the network interface architecture is required instead of modifying many architectural components in comparison to state of the art hardware based methodologies. The overview of state of the art approaches was provided in section 2.3.4.

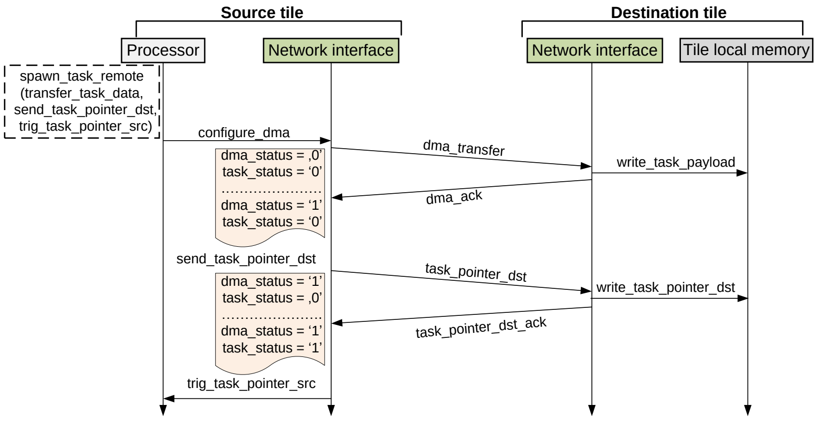


Figure 3.13.: Message sequence chart with network interface managed inter-tile task spawning [145]

For remote task spawning operation, the network interface uses the direct memory access support to move the task data to the destination tile. On the other side, sending of task pointer is the latency sensitive operation which requires the transfer of fixed payload size message. Handling such latency-sensitive communication with conventional network message types reduces performance [149]. Therefore, it is beneficial that task pointers are handled as special network messages in the network interface. In invasive computing, the communication between agent instances to collect resource information and bargain resources represents another use case which benefits from the usage of special network messages. The need for low latency communication with smaller payload size is not necessarily restricted to software-software communication. In state of the art many-core architectures, hardware accelerators assist operating system in task assignment and scheduling duties [116] [136]. The communication between software and such hardware accelerators across the Network on Chip is latency sensitive and has a big impact on the system performance. The example of such communication is the sending of task pointers i.e. i-lets from the operating system to the CiC module in invasive computing.

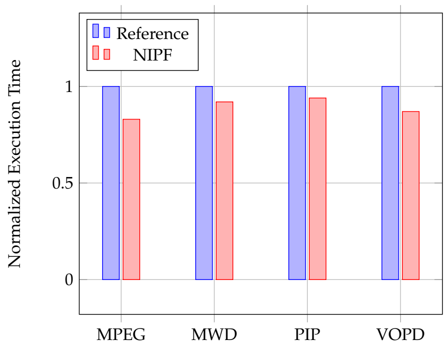
To address the above-mentioned communication requirements, we have introduced the notion of two special message. A system message is the first kind among these message types, which is issued by the network interface to enable fast and low latency communication between software instances. We have named these message as system i-let in our concept. A system i-let is initiated by writing the memory-mapped registers of the network interface. At the receiving tile, an interrupt is issued on the arrival of a system i-let. This mechanism allows an efficient interaction of software instances which are mapped on distributed computation resources by minimizing the effect of latency introduced by the Network on Chip. In order to perform low latency task pointer exchange between different tiles, another special message type is introduced. In the scope of our work, these messages are named as task i-lets.At the sender tile, the sending of task pointer is triggered by writing to the memory-mapped registers of the network interface. At the receiving tile, task i-lets are passed to the hardware accelerator which performs task assignment. In Invasive computing, CiC is the unit which assigns task i-lets to different processing cores keeping in view the application requirements and hardware status [117]. The implementation details of the system-ilet and task-ilet are provided in the section 4.3.

3.2.3. Evaluation using SystemC Modeling and Simulation

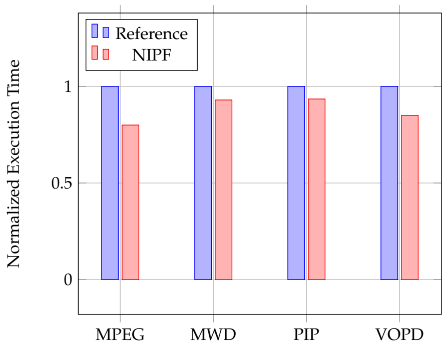
In order to investigate the software communication support, we evaluated the proposed concept for asynchronous data transfer and task spawning. The evaluations are performed in the SystemC based modeling and simulation framework. For the following investigations, only best effort communication mechanism in Network on Chip is used. The modeling of communication constraints and the constraint evaluation performed by the operating system, as discussed in section 3.2, is done at an abstract level. Abstracted behavioral models of processing cores and applications are used for traffic generation. The remaining parameters are kept the same as described in section 3.1.2.

The communication graphs of four multimedia applications, which were introduced in section 3.1.2.2 are used: Video Object Plan Decoding, MPEG4 video decoding, Picture-In-Picture, and Multi-Windows Display. In the first step, hardware support for asynchronous data transfer is evaluated as detailed in section 3.2.1. Each application is executed individually on the simulation platform. The applications use direct memory access hardware support for transferring data between tiles. For the following investigations, the proposed approach in which the network interface is equipped with the proposed data transfer support is referred as NIPF. The configuration without the proposed hardware support is named Reference. Reference configuration also uses the RDMA for data transfer. However, the DMA support in the Reference configuration is not capable of handling the handshaking of data transfer operation between source and destination nodes. The difference between Reference and NIPF comes from the proposed synchronization support for asynchronous data transfer.

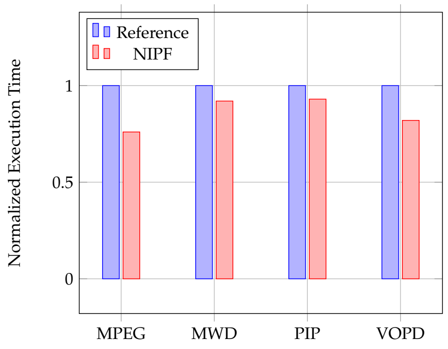
Figure 3.14 shows the normalized execution time of the applications for data transfer support evaluation under different load situations. Varying load situations are created through changing the flit injection rate of the background traffic. Uniform random traffic injected by the nodes of the architecture, which are not occupied by the investigated application, makes the background traffic. Flit injection rate is changed from 0.2 to 0.8 in four different scenarios. As described in section 3.1.2.2, the under-investigation video applications have different communication characteristics. PIP and MWD benefit relatively less from proposed hardware support because of being low bandwidth applications. In comparison MPEG and VOPD show improvement in execution time by a higher amount. The results show that NIPF can improve the execution time by 30% compared to the Reference configuration. The additional latency in the Reference comes from the fact that the software is involved in supervising the status of data transfer operation on both sender and receiver sides. In four different scenarios related to the load changes, it can be observed that the high bandwidth applications show higher performance gain compared to the other applications when the background traffic increases.



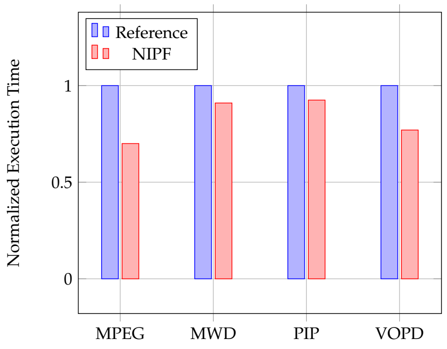
(a) background_traffic_injection_rate = 0.2 Flits/Cycle/Node



(b) background_traffic_injection_rate = 0.4 Flits/Cycle/Node



(c) background_traffic_injection_rate = 0.6 Flits/Cycle/Node

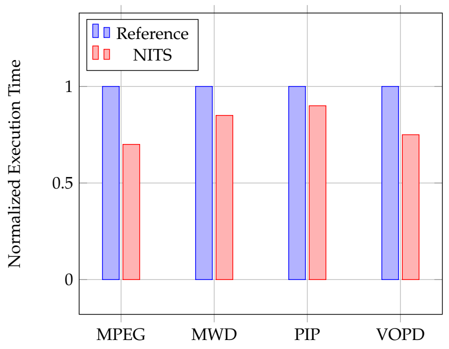


(d) background_traffic_injection_rate = 0.8 Flits/Cycle/Node

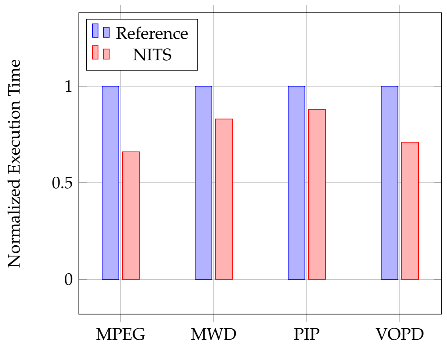
Figure 3.14.: Execution time of real world applications for data prefetching evaluations

In the next step, the task spawning hardware support is evaluated in the simulation framework. Similar to the previous scenarios, each application is executed individually on the platform. The initial number of cores assigned to each application is chosen according to the table 3.1. During execution, each application spawns its computation to twice the number of cores which were initially assigned to it. The applications use task spawning mechanism to spawn their computation as detailed in section 3.2.2. The proposed approach in which the network interface is equipped with task spawning support is referred as NITS. The configuration without the proposed hardware support in the network interface is named Reference. Reference configuration handles the synchronization between software instances for task spawning in software.

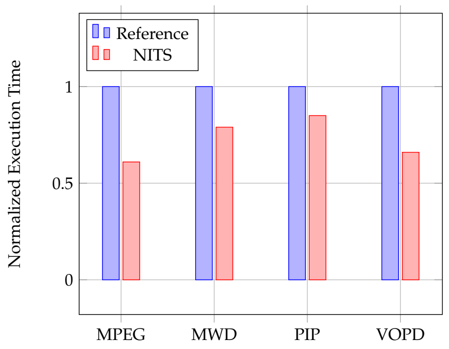
Figure 3.15 shows the normalized execution time of the applications for task spawning under different load situations. Background traffic with varying flit injection rate is generated in four different scenarios. The results depict that NITS improves the execution time by 47% compared to the Reference configuration. MPEG and VOPD benefit from



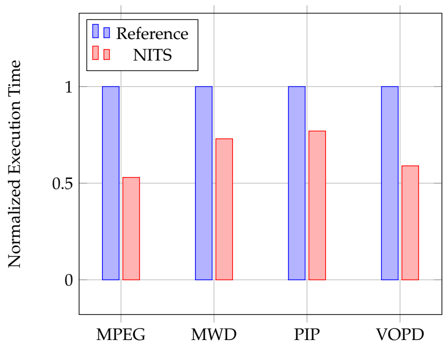
(a) background_traffic_injection_rate = 0.2 Flits/Cycle/Node



(b) background_traffic_injection_rate = 0.4 Flits/Cycle/Node



(c) background_traffic_injection_rate = 0.6 Flits/Cycle/Node



(d) background_traffic_injection_rate = 0.8 Flits/Cycle/Node

Figure 3.15.: Execution time of real world applications for task spawning evaluations

the proposed support by larger proportion because of their higher throughput requirements as compared to other applications. The performance degradation in the Reference configuration comes from the fact that the software is involved in supervising the status of task spawning sub-operations. In four different scenarios for the background traffic variation, it can be observed that the higher throughput applications show more performance improvement compared to the low throughput application when the network load increases.

4. Implementation of Communication Resource Management and Software Communication Support

In this chapter, we have specified the implementation details of the network interface architecture. At first place, the details related to the state of the art NI functionality are described. The state of the art network interface functionality includes features like tile and network protocol translation and Quality of Service support. In later sections, the hardware extensions corresponding to our proposed concepts are narrated. Section 4.2 provides the implementation aspects specific to the communication resource management concept. Hardware realization related to the software communication support is given in the section 4.3. Implementation of the network interface design is carried out in the hardware description language VHDL [101].

4.1. State of the Art Network Interface Components

Since our concepts require extensions in state of the art network interface architecture, we have followed a layered design approach. The layered design approach allows seamless integration of the different features on the top of basic network interface functionality. Network interface architecture is decomposed into following layers; 1) tile interface layer defines the interface with tile interconnect, 2) protocol translation layer performs the tile-network and network-tile translation in transmit and receive datapaths respectively, 3) NI Buffers provide clock domain crossing and isolation of tile interface and NoC interface layers, 4) link interface layer establishes the interface with the Network on Chip. The network interface design with all its functional layers is shown in the figure 4.1.

Concerning the design of the network interface, the above-mentioned layered approach offers following advantages.

- Computation and communication domain isolation which is a big motivation for using a distributed communication infrastructure (Network on Chip)
- Independent refinement of individual network interface design layers
- Support for different NI flavors i.e. network interface for compute tiles, memory and I/O tiles.
- Reduced design complexity
- Transparent utilization of services offered by protocol translation and link interface layers by all modules in the tile interface

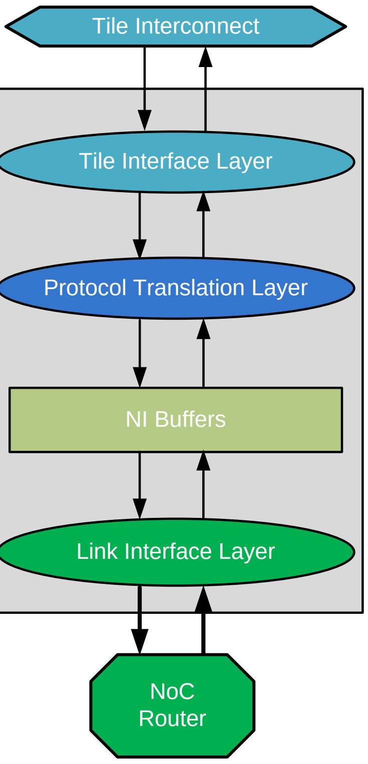


Figure 4.1.: Layered design approach for network interface architecture

Figure 4.2 represents the block diagram of state of the art network interface architecture following the above-mentioned layered approach. In the following section, the implementation details of the components within the individual layers of the network interface architecture are provided.

4.1.1. Tile Interface Layer

Tile interface provides the communication abstraction to the applications running on different tiles. In other words, tile interface acts as a tile bus wrapper. We have divided the tile interface layer into two partitions. Each partition in the tile interface corresponds to a distinct functional unit. One of the functional units is the Remote Load/Store (RLS) unit which services conventional remote shared memory accesses. Whereas the other unit is the Memory-Mapped Registers (MMR) unit. The memory-mapped registers unit consists of sub-modules, which are mapped to different address ranges and support different functionalities. As shown in the figure 4.2, state of the art components in MMR unit are

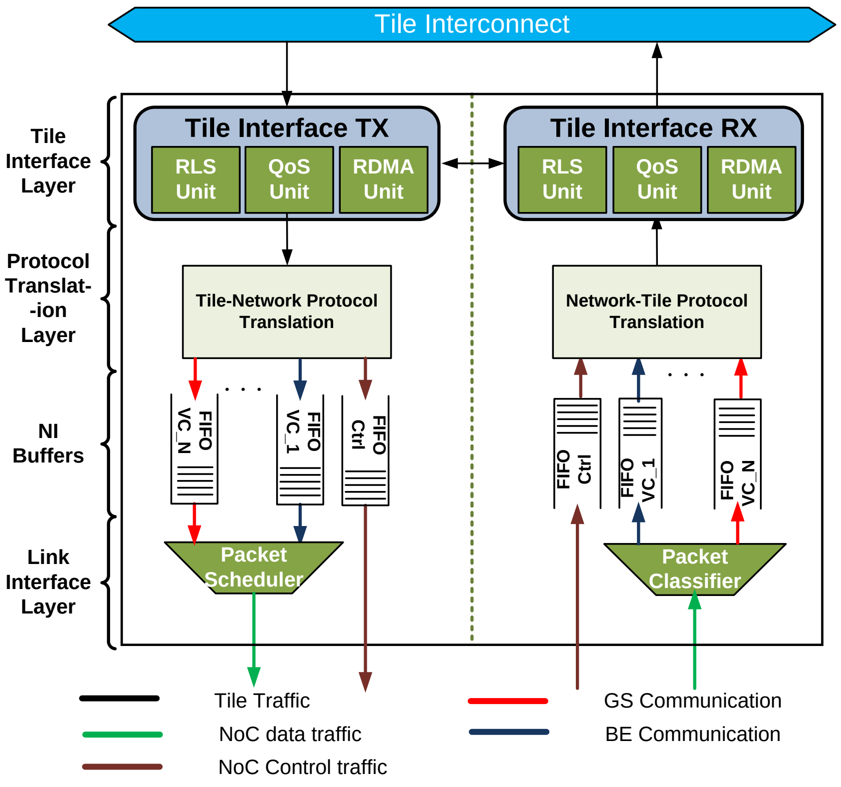


Figure 4.2.: Block diagram of network interface architecture with state of the art communication support

QoS and RDMA units. The block diagram of the network interface architecture with partitioned tile interface layer is shown in the figure 4.3.

Both RLS and MMR units in the tile interface share services in the protocol translation layer of the NI. Each RLS and MMR unit has a distinct interface on the shared bus. Both interfaces are mapped to two different address ranges in the memory map: the shared memory address range and the memory-mapped registers address range. For this purpose, the memory map of the system is divided into the two respective memory access domains. Table 4.1 shows such memory map, which is also deployed in invasive computing. Shared memory address range is further divided into global shared memory and distributed shared memory. The address range for global shared memory is used to access the memory which is shared by all tiles (for example, off-chip DDR memory). The address range of distributed shared memory domain is divided among all tiles to make all tile local memories accessible from each tile. Memory-mapped registers address range contains registers which can be configured to support different services. A separate section is reserved for I/O devices, which are shared between all tiles.

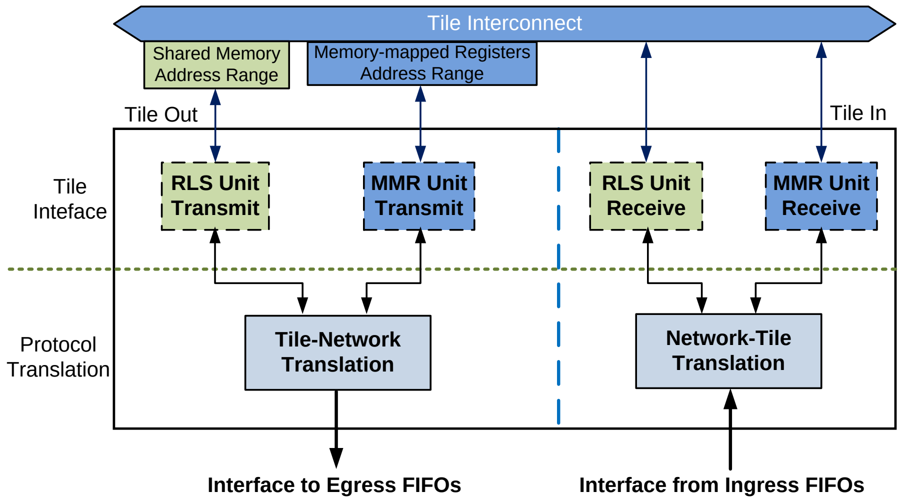


Figure 4.3.: Network interface with partitioned tile interface

Memory access domain		Start Address (MSB)	End Address (MSB)
Global shared memory		0x0000	0x7FFF
Distributed shared memory	Tile 0	0x8000	0x80FF
	Tile 1	0x8100	0x81FF
	Tile...	...	...
Memory-mapped registers		0xC000	0xDFFF
I/O area		0xE000	0xFFFF

Table 4.1.: Memory map realized for distributed shared memory Invasive architecture

The shared memory address range is mapped in a way that it goes through the L2 cache. The shared memory interface of the NI is used for load/store transactions through the L1 and L2 cache hierarchy. It supports read and write transactions to remote tiles transparently. The other interface, which is directly connected to the shared bus, is mapped to the memory-mapped registers address range. The memory-mapped registers interface is used by the software to write into the registers placed in the network interface. An example of the compute tile can be seen in the figure 2.15 where the discussed network interface is incorporated. The details of the micro-architecture of RLS and MMR units is provided in the following text.

4.1.1.1. Remote Load/Store (RLS) Unit

Remote load store unit is the partition in the tile interface which handles all remote shared memory accesses originating from the tile. RLS unit is further divided into RLS Transmit and RLS Receive sub-modules to accomplish the shared memory access support on transmit and receive side datapaths respectively. RLS Transmit module processes remote load/store requests, which are received via tile bus and triggers the tile-network protocol translation. Similarly, the respective receive module is triggered by the network-tile translation whenever a corresponding request arrives in the receive datapath. As shown in the figure 2.15, the network interface receives the physical address for remote access as it is placed after the L2 cache. Virtual to physical address translation is already performed at the Memory Management Unit (MMU) level. Therefore, the functionality of the RLS unit is agnostic to the details of virtual to physical address translation. The hardware implementation of remote shared memory access support is shown in figure 4.4.

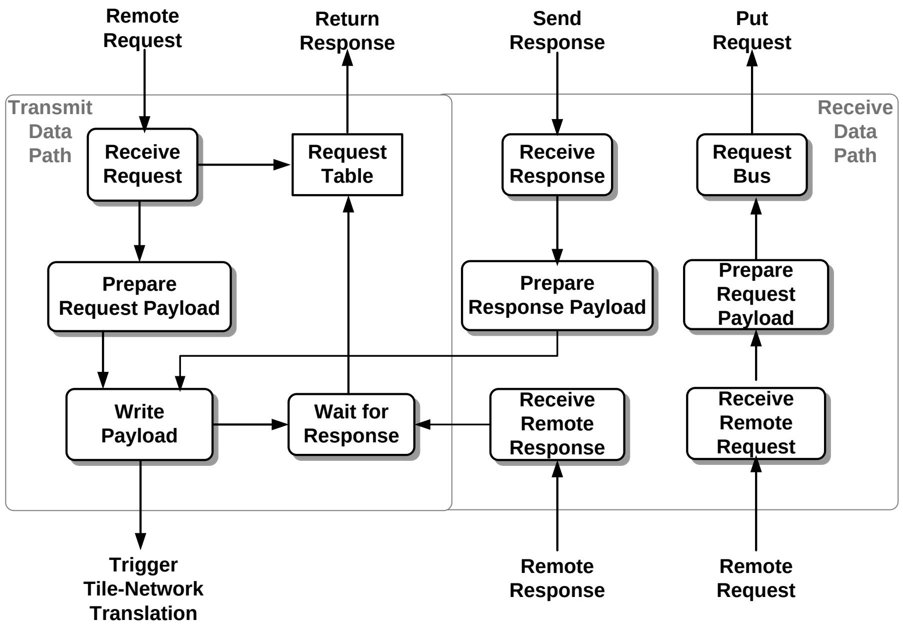


Figure 4.4.: Finite State Machine (FSM) of RLS unit to support multiple outstanding requests

We have implemented remote memory access support for a single word as well as for bursts of configurable size. On the transmit side, remote shared memory requests are received over the bus and stored in a Request Table. Request Table enables serving of multiple outstanding requests through the split transfer AHB bus protocol. Afterward, the request payload is prepared and then forwarded to the tile-network translation to generate the network packet. The network packet corresponding to this request is named as RLS_Req. RLS unit waits for the response of the request from the remote destination tile. However, the tile bus is released according to the split transfer protocol. This brings the benefit that other masters may still utilize the tile bus and thus helps in achieving better throughput. In addition, releasing the tile bus before getting the response becomes essential in the scenarios where message dependencies may result in a deadlock. When the response from the destination tile arrives, it is matched in the Request Table and the corresponding master is signaled to receive the response. On the receive side in the destination tile, network-tile translation forwards remote shared memory requests to RLS Receive unit. After receiving the request, request payload is prepared and a bus request is initiated on the tile bus. When the slave in the destination tile returns the response, response payload is prepared and forwarded to the tile-network translation for generating response packet. This response packet is named RLS_Resp. When the response arrives on the receive datapath of the source tile, network-tile translation forwards it to the transmit side datapath and the response is returned to the master which had initiated the request.

4.1.1.2. Memory-mapped registers (MMR) Unit

Memory-mapped registers unit is the module which contains memory-mapped registers and the associated implementation for supporting the different state of the art services in the network interface. Depending on the functionality, which network interface has to provide, MMR unit is further decomposed into sub-modules. Similar to RLS unit, the corresponding transmit and receive side support for each sub-module is realized. In this section, we have given the details of the MMR sub-modules, which realize state of the art Quality of Service and remote direct memory access support in our design.

Quality of Service (QoS) Unit :

We have used iNoC as the communication infrastructure for our prototype. iNoC supports packet switched communication over virtual channels. Virtual channels can be used for both best effort and guaranteed service traffic types. The data communication which does not require QoS guarantees uses BE traffic. GS traffic is supported through end-to-end connections providing Quality of Service guarantees. End-to-end connections are set up by reserving the virtual channels in the entire communication path. The maximum number of guaranteed service connections which can be established are VC_max-1, where VC_max represents available virtual channels. The reason behind this restriction is that one virtual channel is always left for best effort traffic to prevent complete blocking of communication.

In order to establish a guaranteed service connection, memory-mapped registers inside the network interface need to be configured. Details of the registers, which are required to be written for GS connection setup, are given by the table 4.2. Dst_naddr_reg is used by the tile-network protocol translation layer to encode the network address in the GS head flit. The Quality of Service related parameters of requested guaranteed service connection are expressed by setting Sl_reg register. Once the above-mentioned registers are written, the head flit is issued to establish the GS connection. The head flit travels from the source to the destination tile and performs the virtual channel reservation in the subsequent routers and the network interface units.

Register	Description
Dst_naddr_reg	Register holds 32 bit Destination Network Address of the tile to which GS connection is established
Sl_reg	Register holds the following characteristics of the GS connection Bit 6: Uni/Bidirectional connection (Establish GS connection in both transmit and receive directions) Bit 5: Replaceable connection (Connection can be replaced for self-optimization) Bit 4: 1:Establish connection/0:Release connection Bits 3-0: Service Level which defines the number of time slots which are allocated for scheduling the virtual channel reserved for the connection

Table 4.2.: NI memory-mapped registers for QoS support

Remote Direct Memory Access (RDMA) :

RDMA is the hardware engine realized in the network interface to support the transfer of data between remote tiles. Multiple concurrent DMA operations between tiles are supported by the implemented hardware accelerator. The tile interface of the remote direct memory access unit is shown in the figure 4.5. To initiate a data transfer, memory-mapped registers are configured. Once the required parameters are written into the registers, AHB Transmit Master reads the DMA payload and forwards it to the tile-network translation. In order to avoid the usage of large buffers, the reading of data and the packetization are implemented as pipelined processes. For concurrent DMA operations, RDMA Transmit and Receive Tables are introduced.

In order to initiate a DMA transfer, the memory-mapped registers which need to be configured are shown in the table 4.3. Remote direct memory access Transmit receives the configurations written by the software and stores them in an RDMA Transmit Table. In addition, RDMA Transmit commands the AHB Transmit Master to read payload from Transmit Message Buffer and triggers the tile-network protocol translation to send RDMA payload as network packets. RDMA Receive is the component which triggers AHB Receive Master to write the received payload in the Receive Message Buffer. Transmit and Receive Message Buffers are part of the tile local memory in the transmit and receive tiles respectively.

4.1.2. Protocol Translation

Protocol translation layer is divided between the tile-network translation and network-tile translation modules in the transmit and receive datapaths respectively. Tile-network translation is also referred as Packetization as it converts the requests from tile interface layer to network packets. On the other hand, the network-tile translation is called Depacketization.

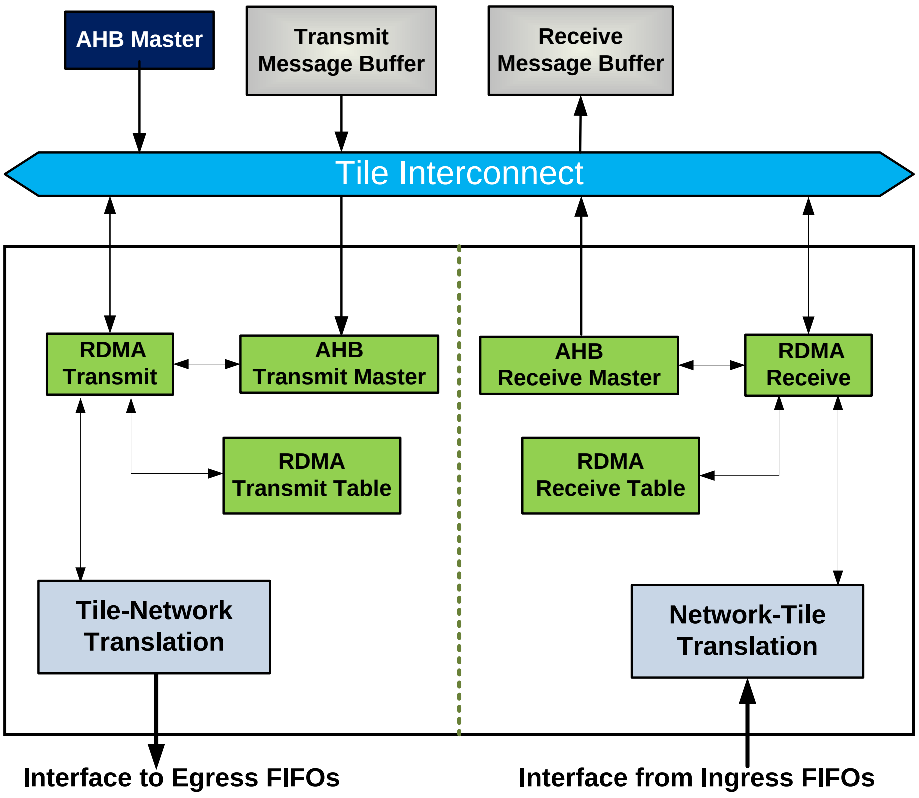


Figure 4.5.: Block diagram of tile interface in RDMA unit

Register	Description
rdma_msg_id_reg	Register holding the unique id corresponding to the remote DMA transfer operation
rdma_payload_length_reg	Amount of data as payload length to be read from Transmit Message Buffer in words
rdma_src_reg	Start address in the tile local memory of the source tile from where the DMA payload has to be read
rdma_dst_reg	Destination address in the tile local memory of the destination tile where the DMA payload has to be written

Table 4.3.: NI memory-mapped registers for data transfer through DMA

4.1.2.1. Tile-Network Protocol Translation

Tile-network translation is responsible for translating the requests from RLS and MMR units and generating network packets from the corresponding requests, which are then forwarded over the Network on Chip. AMBA AHB 2.0 is used as the tile interconnect in the invasive computing [126]. In our case, iNoC is used as the Network on Chip, which is the virtual channel based network supporting both guaranteed service and best effort communication [57]. The details about the iNoC are already provided in section 2.4.3.2.

The network packet corresponding to best effort communication is shown in the figure 4.6. In best effort traffic, the communication between the source and destination happens in the form of packets. Each packet contains head and tail flits in addition to the payload flits corresponding to the specific request. In order to ensure transparent utilization of protocol translation layer by the modules in the tile interface layer, the corresponding request type is encoded in a special flit. This flit is named as *request type* flit in our implementation. RLS_Req and RLS_Resp are the possible request types for remote shared memory access. Whereas, GS_Req and RDMA_Req are the corresponding request types for GS connection and remote DMA transfer. An additional flit is used to encode the request type to transparently use the tile interface layer by the protocol translation layer for both guaranteed service and best effort communication. In addition, the request type flit contains the number of payload flits which follow it. The number of payload flits may vary depending on the request type. The head flit carries the routing information for the packet in the network. Whereas the tail flit indicates the end of the packet.

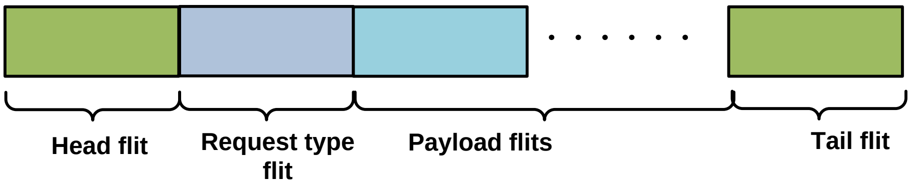


Figure 4.6.: Network packet corresponding to connection-less best effort communication

Guaranteed service communication is shown in the figure 4.7. Guaranteed service communication is realized through end-to-end virtual channel based connections. For GS transmission, head and tail flits mark the start and end of the whole communication. After the successful reservation, payload flits can be sent from the source to the destination without any overhead flits. In this case, the overhead flits are the head and tail flits which would have been present in each packet in case of best effort communication. The GS connection is established by sending a head flit from the source network interface to the destination network interface. The head flit performs the reservation of virtual channels in the source and destination network interface units as well as in the routers, which exist in the path. Once the head flit is received by the destination tile and the reservation of virtual channels is successfully done, the acknowledgment flit is sent. Once the connection has been established, it can be used transparently by all messages which are routed to the same destination tile. GS connection is terminated by sending a tail flit which is responsible for releasing the resources in the network interface and the subsequent routers.

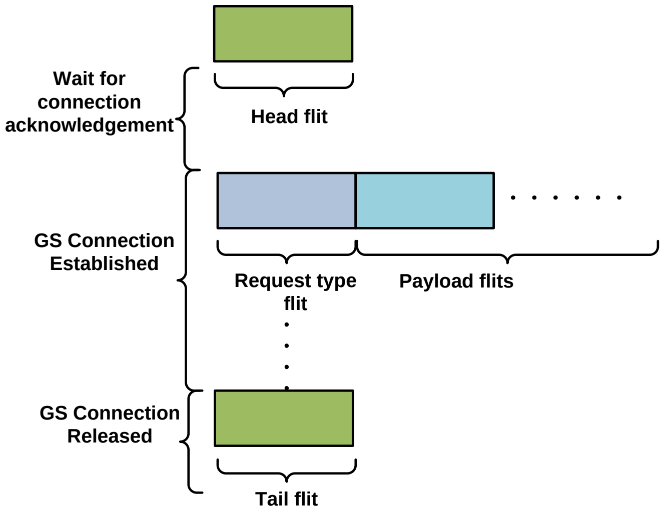


Figure 4.7.: Connection oriented guaranteed service communication

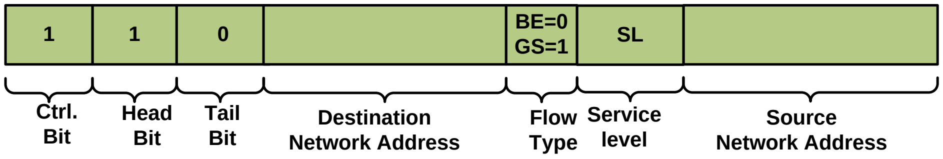


Figure 4.8.: Head flit format

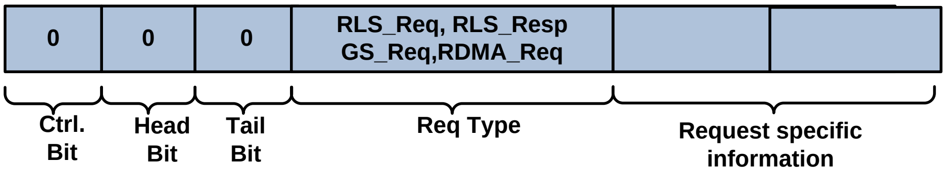

0
0
0
RLS_Req, RLS_Resp GS_Req,RDMA_Req
Ctrl. Bit
Head Bit
Tail Bit
Req Type
Request specific information

Figure 4.9.: Request type flit format

The formats of the head, request type and tail flits are shown in the figure 4.8, 4.9 and 4.10 respectively. Head flit indicates the start of transmission. The head flit is 33 bits wide assuming 32 bit datapath width of Network on Chip. The most significant bit in the head flit is named *Ctrl* bit. The Ctrl bit is set to '1' when the network interface sends

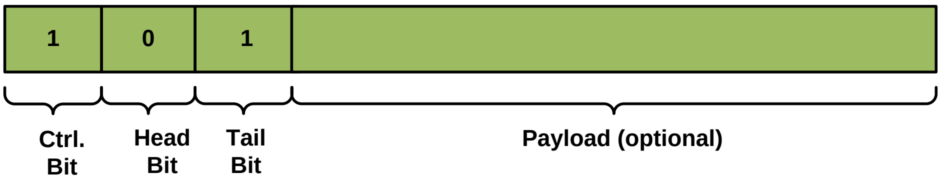


Figure 4.10.: Tail flit format

head and tail flits to the router. Otherwise, this bit is kept '0', during the transmission of request type and payload flits. In the head flit, Destination and Source Network Address fields specify the network id of the source and destination tiles respectively which are important for guiding the packet over Network on Chip. Request type flits are interpreted by the destination tile network interface. These flits contain *Req Type* field which is used to uniquely encode different request types (like RLS_Req and RLS_Resp) so that they can be classified in the receive datapath. In addition, this flit contains request specific information which is important to process the request on the receiver side. For example, the total number of payload flits contained in the message are also specified, which may vary in the case of remote direct memory access request. The tail flit is processed by the router to acknowledge the end of the current packet in best effort communication and the release of connection in case of GS transmission. Tail flits may also contain the payload.

In our implementation, Packetization is pipelined and is triggered immediately when it is requested by the tile interface layer. Therefore, it does not produce additional performance overhead.

4.1.2.2. Network-Tile Protocol Translation

On the receive side, network-tile translation processes the incoming requests from the network which are temporarily stored in the network interface buffers in receive data path. Network-tile translation interprets the Req Type field in the request type flit and triggers RLS and MMR units in the receive datapath accordingly. In addition, the id of the sender of the request is stored in case the reply of the message has to be sent.

4.1.3. NI Buffers

Network interface buffers are realized by keeping in view the following design constraints.

- Virtual channel based packet switched communication requires the independent storage of packets inside the network interface. The packets may correspond to a particular traffic flow i.e. GS/BE communication or a specific message type i.e. remote shared memory access/RDMA.
- A mechanism is required to provide clock domain separation between computation (tiles) and communication (Network on Chip).

- Back pressure management within the network interface for tile and network interface layers.

Therefore, First in First out (FIFO) memories are incorporated in the network interface datapath to achieve the above-mentioned design objectives.

Each FIFO realizes an independent virtual channel. The network interface and the router architecture support dynamic sharing of virtual channels between guaranteed service and best effort traffic. In other words, each virtual channel in the Network on Chip can either be used by GS or BE traffic respectively. The virtual channels which are not occupied by the guaranteed service traffic can be used by best effort communication. However, there is at least one virtual channel which is always kept for BE communication in order to avoid reservation of all virtual channels by the guaranteed service traffic. This restriction on the number of GS connections prevents the complete starvation of best effort communication.

If a GS channel exists to a particular destination tile, a FIFO/virtual channel is classified to store the requests which are sent to that tile only. However, in the case of best effort communication, FIFOs are not reserved to store packets which are sent for a particular destination tile. In addition, accesses corresponding to different request types may share the same FIFO. For example, remote load/store requests and RDMA can share the same FIFO in the network interface. As the name suggests, the output of the FIFO always carries the oldest flit in the buffer. In order to provide clock domain separation, FIFO can be read and written at different clock frequencies. Although, in our current implementation, tile interconnect and the Network on Chip are clocked at the same frequency. In the transmit datapath, packets are written into the FIFOs from tile interface layer and read from the link interface layer. In the receive datapath, roles of tile interface and link interface are reversed while accessing the FIFOs.

4.1.4. Link Interface Layer

Link interface layer establishes the interface of protocol translation layer to the Network on Chip. iNoC is the on-chip network which we have deployed for connecting different tiles [56]. Wormhole switching is used by the iNoC to minimize the buffering requirements. iNoC offers two communication channels which represent orthogonal physical networks; a data channel and a control channel. The data channel is used for conventional data traffic including both guaranteed service and best effort communication. Whereas control channel represents another physical network layer which is used for delivering network internal messages. For example, the acknowledgment flit for GS channel establishment is routed over the control channel. It is done to avoid message-dependent deadlocks and reduce delays in GS connection establishment. In addition, the control channel is used for transferring the information related to the Network on Chip utilization, which can be used by software layers to perform application mapping [137].

For data channel, the link interface can be further divided into packet scheduling and packet classification modules, keeping in view its functionality in transmit and receive datapaths respectively.

4.1.4.1. Packet Scheduling

Packet scheduler module is responsible for sending data from transmit virtual channels in the network interface datapath to the output link. Therefore, it has to perform two functions; 1) scheduling of the virtual channel over the output link keeping in view the availability of data and Quality of Service requirements 2) establishment of the flow control between the network interface and the neighboring router for data transmission.

Scheduling over the output link is carried out in a scheduling cycle, which is defined by the equation 4.1.

$$Cycle_{sch} = \sum VC\_Timeslots_i \tag{4.1}$$

Where $Cycle_{sch}$ represents the scheduling cycle and $VC\_Timeslots_i$ defines the number of time slots, which are assigned to the virtual channel i. The total number of time slots define the bandwidth over the output link as given by the equation 4.2.

$$Link_{bw} = \sum Timeslot_j \times data_j \tag{4.2}$$

Where $data_j$ represents the data which is sent over the link in the $Timeslot_j$. $Link_{bw}$ defines the link bandwidth. Each virtual channel is allocated the number of time slots in the scheduling cycle according to its QoS requirements. Each virtual channel, which is being used for the best effort communication, is allocated a single time slot. In the case of guaranteed service virtual channel, the number of allocated time slots are defined in the service level register when the connection is being initiated.

Weighted round robin scheduling as detailed in section 2.2.5 is deployed as a scheduling strategy in the transmit datapath of the network interface. This scheduling policy ensures fairness between virtual channels and enforces Quality of Service requirements. Each virtual channel is served by the output link for the number of time slots which are allocated to it. A virtual channel is a candidate to be scheduled over the link if it has data to be transmitted and the receiving buffer is ready to accept data. The scheduling of only candidate virtual channels over the output link ensures the maximum throughput in comparison to a TDM-based scheduling. The availability of data is indicated, when the FIFO in the transmit datapath which corresponds to the virtual channel is not empty. Credit-based flow control is used to ensure availability of buffer space in the subsequent router. As described in section 2.2.6, credit-based flow control makes it possible to pipeline the links between the routers without significant performance degradation. Credit counters are implemented in the packet scheduler, which represent the available credits for the respective virtual channels.

4.1.4.2. Packet Classification

Incoming data from the Network on Chip in the receive datapath is handled by the packet classifier. It receives incoming data from the Network on Chip and stores them in the respective receive datapath FIFOs. Similar to the transmit datapath, credit-based flow control is implemented to store the incoming data. The scheduling of FIFOs in the receive datapath is done by using a round robin arbitration policy. The granularity of scheduling on the receive side is the complete incoming request instead of flits. WRR scheduling policy can not be implemented for the receive side virtual channels because the complete request has to be served over the bus.

Figure 4.11 represents the block diagram of the network interface architecture including the hardware extensions which correspond to our concepts. AUTO_GS concept requires

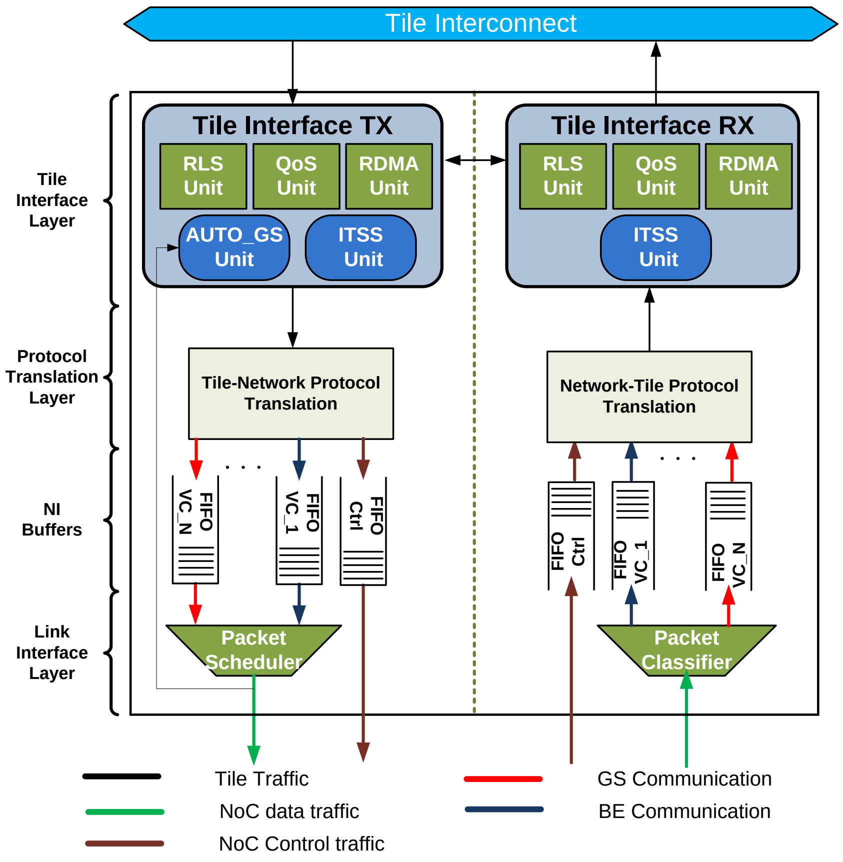


Figure 4.11.: Block diagram of proposed network interface architecture

modifications only in the transmit datapath of the network interface and are represented by the AUTO_GS unit in the figure 4.11. The inter-tile synchronization support is highlighted through the ITSS unit in both transmit and receive datapaths. The implementation details of the proposed extensions will follow in the next sections.

4.2. Hardware-controlled GS Connections (AUTO_GS)

According to the concept of hardware-controlled communication resource management, presented in section 3.1.1, AUTO_GS methodology establishes guaranteed service connections keeping in view dynamic communication patterns. Therefore, the network interface architecture is extended with traffic monitoring and autonomous GS connection management capabilities.

The block diagram of the network interface transmit datapath with the AUTO_GS hardware support is shown in the figure 4.12. The figure shows the combination of both

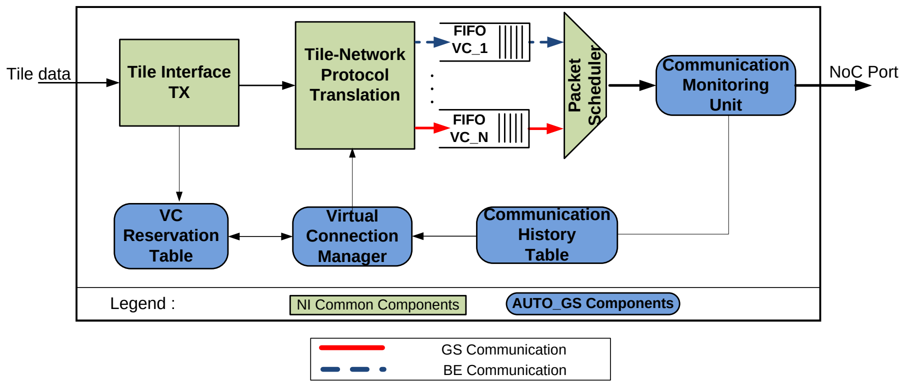


Figure 4.12.: Network interface transmit datapath block diagram with AUTO_GS support

state of the art components and the introduced hardware extensions corresponding to the AUTO_GS concept. Tile interface TX checks for the existence of a reserved connection to the destination tile in each incoming request. In both cases, the tile-network protocol translation unit is triggered to generate flits or packets for connection-based or connectionless traffic respectively. Afterward, the data is placed in the respective FIFOs. The packet scheduler is responsible for scheduling the different FIFOs over the network interface output link. Implementation details specific to the above-mentioned state of the art components are already given in the previous sections.

Among the proposed extensions, the *VC Reservation Table* contains the identifiers (ids) of destination tiles to which a virtual channel based end-to-end connection exists. The packets which are being sent out over the network are monitored by the *Communication Monitoring Unit*. The *Communication History Table* holds the communication history of the tile in the form of a list, which contains the destination tiles to whom the packets are sent out. The list is sorted in the order of increasing number of packets. The *Virtual Connection Manager* is the module which ensures the establishment/replacement of hardware-controlled GS connections. The following sub-sections provide a detailed description of the modules which are implemented to support the AUTO_GS concept.

4.2.1. Virtual Channel Reservation Table :

In the network interface architecture, the status information of the tile's virtual channels is stored in VC Reservation Table. VC Reservation Table can be accessed by the software through memory-mapped registers for QoS communication. For the sake of AUTO_GS concept, the entries in the VC Reservation Table are also made accessible to the Virtual Connection Manager hardware block. Table 4.4 provides the details of the registers which can be accessed for inquiring the existence of a reserved connection to a given destination tile.

Register	Description
vc_dst_reg	Register holds 32 bit Destination Network Address of the tile to which the existence of a GS connection is inquired
vc_dst_status_reg	Read only register which returns the VC status corresponding to destination tile written in vc_dst_reg.

Table 4.4.: NI memory-mapped registers showing existence of VC based connection to a given tile

In addition, the reservation status of an individual virtual channel can be read through the status registers, which are given in the table 4.5. Table 4.5 has entries equal to the number of available virtual channels. Depending on the status of the available VCs and

Register	Description
vc0_status_reg	Register holding the status of the virtual channel number 0
vc1_status_reg	Register holding the status of the virtual channel number 1
...	...

Table 4.5.: NI memory-mapped registers showing the reservation status of individual virtual channels

the on-going guaranteed service connection reservation, virtual channel can have one of the following statuses:

- **VC_Free :** Virtual channel is not reserved for any GS connection and can be used for establishing a new connection or for best effort communication.
- **RSV_ONGOING :** GS connection establishment is on-going and the end-to-end connection is not yet acknowledged.
- **RSV_SUCCESS :** GS connection is successfully established.
- **RSV_FAILED :** GS connection is failed because of unavailable of virtual channels.

4.2.2. Communication Monitoring Unit

The Communication Monitoring Unit (CMU) monitors the data which is sent out over the network by the packet scheduler. In the case of best effort communication, the head flit of each packet is monitored. For tracking the utilization of existing hardware-controlled GS connections, the packet count is incremented when the number of transmitted flits equals to best effort packet size. Initially, no GS connections are existing and the entire traffic is served via best effort communication. CMU checks the Destination Network Address field contained in the head flit of each outgoing packet and tries to find it in the Communication History Table. If the Destination Network Address is not found in the Communication History Table i.e. the packet is being sent to a destination tile for the first time, it is registered in the table as a new entry. The number of entries in the Communication History Table is restricted by the parameter AUTO_GS$_{conn}$. However, if the entry corresponding to Destination Network Address already exists in the table, the packet count for this entry is incremented. In this way, the Communication Monitoring Unit is responsible for maintaining the communication history of the tile in the network interface. Communication history is gathered for a configurable time interval AUTO_GS$_{cycle}$. After AUTO_GS$_{cycle}$ is passed, Communication Monitoring Unit stops collecting the history and triggers the Communication History Table to start analyzing the communication history. Afterward, Communication Monitoring Unit waits for the signal AUTO_GS_SAMPLE from Virtual Connection Manager to start monitoring again.

4.2.3. Communication History Table

Communication History Table (CHT) maintains the packet count field for all the tiles to whom the data is sent from the current tile. CHT starts sorting its entries upon receiving AUTO_GS_SORT signal from Communication Monitoring Unit. A simple sorting algorithm is implemented in a sequential manner to reduce the implementation cost of hardware. Entries to be sorted are stored in a RAM. A control FSM is implemented which loops over all RAM locations and performs the sorting. The entries are sorted in the order of high packet count value. It means that the tile to which the most packets are sent is placed at the top of the table. While performing the sorting, the entries are also evaluated for their qualification for GS connection. If the entry qualifies for a GS connection, it is marked as a hotspot by setting the Hotspot field to '1'. Otherwise, it is marked as coldspot i.e. Hotspot field set to '0'. After the Communication History Table is finished with the sorting, it triggers Virtual Connection Manager with AUTO_GS_CONNECT signal to start establishing GS connections according to the sorted entries.

4.2.4. Virtual Connection Manager

Virtual Connection Manager (VCM) is the module which is responsible for managing the hardware-controlled GS connections when triggered by Communication History Table. In the first step, each entry in the Communication History Table is evaluated, if a GS connection to this particular target tile already exists. This is done by checking the target tile id in the VC Reservation Table. If the entry already exists in the VC Reservation Table, Virtual Connection Manager skips it and proceeds further to process next entry in the Communication History Table. However, if a hotspot entry is encountered to which a connection is not found in VC Reservation Table, VCM triggers the tile-network translation for generating a head flit which establishes the GS connection to the target tile.

The situation may occur where a hotspot entry is still left in the Communication History Table but the number of allowed connections are already established. In this case, the Virtual Connection Manager searches for the previously established GS connections to cold-spots. In the first step, Virtual Connection Manager checks for the existence of a GS connection to a cold-spot entry in VC Reservation Table. In the case where GS connections exist to more than one cold-spots, the one with the least packet count value is chosen for replacement. Afterward, the tile-network protocol translation is triggered to issue tail flit for the corresponding cold-spot entry. If all available GS connections are assigned to hotspots but still there are hotspot entries left in the Communication History Table, those are skipped over by the Virtual Connection Manager. When the Virtual Connection Manager is finished with the management of guaranteed service connections, Communication Monitoring Unit is signaled to start collecting the communication history again. The time to analyze communication history and AUTO_GS connection setup is minimal because of being a hardware implementation. However, the optimum performance of the AUTO_GS concept in a given scenario depends on the appropriate selection of monitoring interval i.e. $AUTO\_GS_{cycle}$. The interplay of hardware modules to implement the AUTO_GS concept is shown in the figure 4.13.

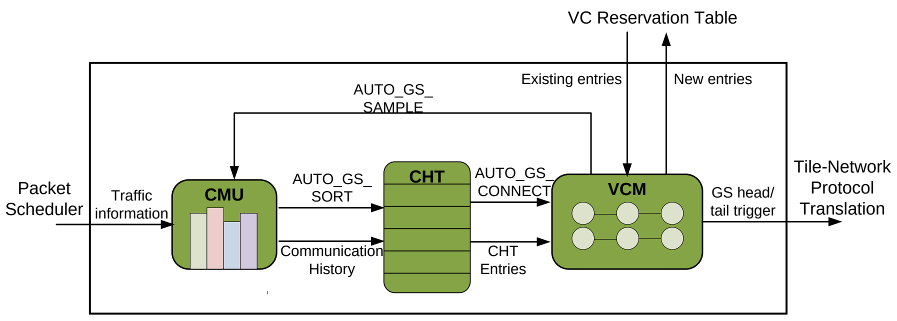


Figure 4.13.: Interplay of hardware modules for AUTO_GS concept

4.2.5. Synthesis Results

The network interface with the AUTO_GS components has been implemented using VHDL. An HDL implementation is required to obtain accurate numbers for resource requirements, power consumption, and the achievable clock frequency. These numbers are then important to compare different implementation alternatives accurately with respect to energy and resource efficiency. Hardware extensions corresponding to AUTO_GS concept can be enabled and disabled through a parameter at design time. In addition, the number of monitored connections $AUTO\_GS_{conn}$ and monitoring interval $AUTO\_GS_{cycle}$ parameters can be configured before synthesis.

The implementation cost of the introduced hardware extensions is measured in terms of area and clock frequency. The network interface used for synthesis has the same features and characteristics as the one which was used for SystemC based simulation framework. A Xilinx Virtex-5 VLX330 FPGA is used as a target prototyping chip. Synopsys Synplify Premier G-2012.09 is used for synthesis. After the synthesis, place and route is performed by Xilinx P&R tools. Table 4.6 shows the synthesis results of a single network interface unit after P&R with $AUTO\_GS_{conn}$ and $AUTO\_GS_{cycle}$ set to 2 and 4160 respectively.

Synthesized Entity	**ASIC Area** (μm^2)	**FPGA**	
		LUTs	**Registers**
NI (without AUTO_GS Extensions)	43579	4937	1868
Communication monitoring unit	1541	102	61
Communication history table	1663	121	58
Virtual connection manager	932	113	91

Table 4.6.: ASIC TSMC 45 nm and FPGA Virtex-5 VLX330 synthesis Results with $AUTO\_GS_{conn}$ = 2, $AUTO\_GS_{cycle}$ = 4160

Compared to the NI with basic functionality, the version with proposed extensions requires only 336 LUTs and 210 Register bits. The synthesis results show that the proposed hardware extensions have a very small footprint at the target FPGA device (< 1%). The similar observation for area consumption is also valid regarding ASIC synthesis. In addition, the introduced hardware support brings small area overhead when compared to the network interface with basic functionality. For Basic NI, the clock frequency of up to 183 MHz could be achieved on the FPGA platform. The clock frequency is decreased to 174MHz with the hardware extensions. This is mainly due to the increased critical path length inside the network interface because of the Communication Monitoring Unit. On ASIC platform, the network interface could be synthesized with a target frequency of 1500 MHz. The network interface with proposed hardware modules brings the maximum achieved frequency down to 1425 MHz.

In order to observe the impact of AUTO_GS parameters on implementation cost, we performed further investigations with different $AUTO\_GS_{conn}$ and $AUTO\_GS_{cycle}$ values. Table 4.7 shows the resource utilization of AUTO_GS hardware modules with a varying number of monitored connections. The area consumption results for changing values of monitoring interval are provided by the table 4.8. In general, increasing either of $AUTO\_GS_{conn}$ and $AUTO\_GS_{cycle}$ raises the area cost of AUTO_GS implementation. Communication History Table is directly influenced by $AUTO\_GS_{conn}$ and $AUTO\_GS_{cycle}$ parameters. Setting the number of monitored connections or monitoring interval to bigger value results in higher logic element utilization in Communication History Table module. Communication History Table uses RAM for storing the communication history. For ASIC synthesis, the memory is always implemented using logic cells. On the FPGA platform, the RAM is realized using Lookup Tables (LUTs) as the default implementation. In order to save logic elements in FPGA, another possible implementation could be chosen at design time which uses BRAM blocks for storing communication history. AUTO_GS hardware synthesized with either using logic elements or BRAMs is represented with LE and BRAM suffixes in the tables 4.7 and 4.8 respectively. However, synthesizing with BRAM option reduces the maximum achievable frequency on FPGA platform to 150 MHz.

The implementation cost of Communication Monitoring Unit and Virtual Connection Manager is not affected from the $AUTO\_GS_{cycle}$ value. The Communication Monitoring Unit is agnostic to monitoring interval because it samples the communication statistics on a packet basis. Similarly, the functionality of Virtual Connection Manager is not dependent on $AUTO\_GS_{cycle}$ as it is explicitly triggered by Communication History Table through AUTO_GS_CONNECT signal. However, the higher $AUTO\_GS_{conn}$ value makes the area cost of both of these modules slightly larger. This is due to the fact that the higher amount of logic elements are consumed when more number of connections are to be monitored.

Synthesized Entity	ASIC Area (μm^2)	FPGA LUTs	FPGA Registers	FPGA BRAM
AUTO_GS_LE ($AUTO\_GS_{conn}$ = 4)	4456	360	225	-
AUTO_GS_LE ($AUTO\_GS_{conn}$ = 6)	5090	403	248	-
AUTO_GS_LE ($AUTO\_GS_{conn}$ = 8)	5965	469	267	-
AUTO_GS_BRAM ($AUTO\_GS_{conn}$ = 8)	5965	242	267	2

Table 4.7.: AUTO_GS synthesis results with different $AUTO\_GS_{conn}$ values and $AUTO\_GS_{cycle}$ = 4160

Synthesized Entity	ASIC Area (μm^2)	FPGA LUTs	FPGA Registers	FPGA BRAM
AUTO_GS_LE ($AUTO\_GS_{cycle}$ = 8320)	4310	350	210	-
AUTO_GS_LE ($AUTO\_GS_{cycle}$ = 12480)	4596	371	210	-
AUTO_GS_LE ($AUTO\_GS_{cycle}$ = 16640)	4904	395	210	-
AUTO_GS_BRAM ($AUTO\_GS_{cycle}$ = 16640)	4904	226	210	2

Table 4.8.: AUTO_GS synthesis results with different $AUTO\_GS_{cycle}$ values and $AUTO\_GS_{conn}$ = 2

4.3. Inter-tile Software Synchronization Support

As stated in section 3.2, we have considered the asynchronous data transfer and remote task spawning scenarios for investigating the synchronization support for software layers. Hardware support for inter-tile synchronization was represented by the ITSS unit in the figure 4.11. ITSS unit can be further divided into RDMA Signaling, System i-let Generation and Task i-let Generation units. The implementation work for the above-mentioned hardware blocks is described in the following text.

4.3.1. RDMA Signaling

The prefetching concept to use remote direct memory access in the network interface is already discussed in the chapter 3. The important characteristic, which is introduced in the RDMA hardware, is the support for the efficient signaling mechanism.

In order to offload the software from polling status of data transfer operations, it is essential that the RDMA unit is able to handle the bookkeeping for data transfer in hardware. Otherwise, the software has to poll the status of previously initiated DMA operations. Remote data transfer operation is done using state of the art RDMA unit in the network interface. However, in order to signal the DMA completion, RDMA Signaling in the receive datapath initiates a special packet to the source tile which indicates the end of the DMA operation on the receive side. Upon the reception of DMA completion message on the source tile, RDMA Signaling support issues a maskable interrupt to inform the software. Table 4.9 illustrates the registers, which tell about the completion of RDMA operation on both source and destination tiles.

Register	Description
rdma_src_status_reg	Register indicating the completion of data copy operation on the source tile
rdma_recv_status_reg	Register indicating the completion of RDMA operation on the source tile after receiving the DMA completion packet from destination tile

Table 4.9.: NI memory-mapped registers for DMA Status

4.3.2. System i-let Generation

In order to support efficient communication between higher layer software instances which are running on remote tiles, we have introduced special network messages named as system-ilets. Table 4.10 shows the registers which are required to be configured for sending the system i-let. The number of system i-let payload registers is currently fixed to 8. Upon writing the last payload register, the tile-network protocol translation is triggered for sending the system i-let towards the destination tile.

Upon receiving system i-let on the receive side, the network interface sends an interrupt to a processing core for immediately serving the request contained in the system i-let. The system i-let payload is written at the memory location pointed by the operating system. The processing of system i-let in the receive datapath is done in a blocking manner i.e. network interface can only proceed for serving the next system i-lets when the operating

Register	Description
sys_ilet_src_status_reg	Status register indicating the flow control for system i-lets '0': System i-let transmission possible at source side '1': System i-let transmission not possible at source side
sys_ilet_dst_id_reg	Address Pointer in the destination tile, where the system i-let has to be sent
sys_ilet_payload_reg1	System i-let payload register 1
...	...

Table 4.10.: NI memory-mapped registers for sending system i-let at source tile

system is ready to process it. The memory-mapped registers, which are made available in the network interface receive datapath for processing a system i-let, are shown in table 4.11.

Register	Description
sys_ilet_dst_status_reg	This register contains the status of the system i-let operation on the receive side. Network interface sets a flag in the register and then polls it till the operating system clears the flag to indicate the completion of system i-let processing
sys_ilet_dst_ptr_reg	Register containing the start address in the tile local memory, where the current system i-let payload has to be written

Table 4.11.: NI memory-mapped registers for processing system i-let at destination tile

4.3.3. Task i-let Generation

In order to exchange task pointers between applications running on different tiles, special network messages are used which are named as task i-lets. In the scope of invasive computing, the memory-mapped registers inside the network interface must be configured to send the task i-let. The memory-mapped registers for sending task i-let are shown in the table 4.12. Upon receiving the task i-let, the network interface writes the task i-let payload in the CiC. CiC is responsible for assigning tasks to different processing cores in the tile. The number of payload registers in a task i-let is currently fixed to 4.

Figure 4.14 shows the transmit datapath of the network interface with the proposed hardware extensions. RDMA Signaling, System i-let Generation and Task i-let Generation represent the hardware modules for software communication support. Figure 4.15 shows the

Register	Description
task_ilet_dir_status_reg	Status register indicating the flow control for task i-lets '0': Task i-let transmission possible at source side '1': Task i-let transmission not possible at source side
task_ilet_dir_msg_id_reg	Message identifier indicating unique id of the task pointer
task_ilet_dir_payload_reg1	Task i-let payload register 1
...	...

Table 4.12.: NI memory-mapped registers for sending task i-let

corresponding software communication support in the receive datapath of the network interface.

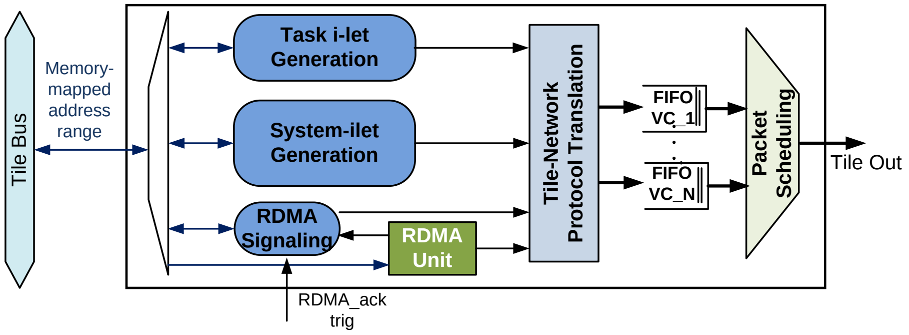


Figure 4.14.: Network interface transmit datapath with software communication support

Keeping in view the case studies of asynchronous data transfer and remote task spawning, the interplay of the above-mentioned hardware components is important. For asynchronous data transfer, the software instances communicate through system i-lets to acquire the data pointer of the remote memory location, where the data has to be copied. Afterward, the remote direct memory access is initiated by configuring the state of the art RDMA unit in the network interface. RDMA Signaling unit is responsible for managing the status of the DMA operation. The software is informed about the completion of the DMA operation through a maskable interrupt. In this manner, software is relieved from polling the status of remote data transfer operation. Therefore, the proposed hardware support assists in realizing the data prefetching by offloading the software as detailed in the section 3.2.1.

For remote task spawning, the software on the sender side configures the network interface with the required DMA data pointers and task pointer in the start. Afterward, the network interface triggers the RDMA unit and the Task i-let Generation units internally to perform different sub-operations, which are required in the remote task spawning as detailed in section 3.2.2. The bookkeeping of sub-operations is done in the network interface internally. Therefore, the software is offloaded from handling the synchronization which is required in the remote task spawning operation. The experiments and the corresponding results of the task spawning case study are presented in the section 5.3.

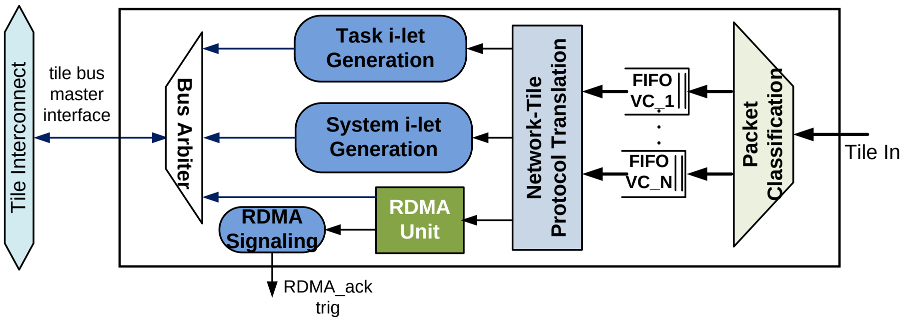


Figure 4.15.: Network interface receive datapath with software communication support

4.3.4. Synthesis Results

Table 4.13 shows the resource utilization of a single network interface with and without proposed software communication support for both ASIC and FPGA platforms respectively. For ASIC synthesis, a 45 nm standard cell library from TSMC (tcbn45gsbwpwc)

Synthesized Entity	ASIC Area (μm^2)	FPGA LUTs	FPGA Registers
NI (without Software Communication Support)	43579	4937	1868
RDMA Signaling	1784	181	86
System-ilet Generation	3121	408	174
Task-ilet Generation	2235	309	109

Table 4.13.: ASIC TSMC 45 nm and FPGA Virtex-5 VLX330 synthesis Results

with worst case operating conditions is used. Synopsys Design Compiler (F-2011.09-SP4) is taken for ASIC synthesis. The target device for the FPGA synthesis is Xilinx Virtex-5 VLX330 which is also used for prototyping. Synopsys Synplify Premier (G-2012.09) is used for FPGA synthesis. After the synthesis, place and route for FPGA is done by Xilinx P&R tools. The results depict that the proposed hardware support for software communication offers a low area footprint. The area overhead of communication support is around 16% for both ASIC and FPGA implementations in comparison to the network interface without corresponding support. In addition, the proposed hardware can be synthesized with the maximum achievable frequency of 1500 MHz for ASIC implementation which is comparable to the interconnect frequency of the state of the art many-core architectures like SCC [67].

5. Experimental Setup and Validation

In this chapter, the details about the RTL simulation framework will be provided, which was used for the investigation of the proposed concepts in the network interface. FPGA prototypes which are built to explore the performance improvements offered by our network interface enhancements are also described. In addition, the results related to the proposed extensions would be provided and discussed.

5.1. Cycle Accurate Simulation Framework

A cycle accurate RTL simulation framework is built to investigate the proposed concepts. An RTL implementation helps to evaluate the proposed concepts at the suitable design abstraction as all system components including software are present to have fair performance comparison. RTL simulation framework enables the investigation of the scenarios which could not be realized on the FPGA prototype because of the implementation cost. The network interface is implemented in VHDL whereas the NoC router is realized in SystemVerilog [102]. Simulation framework supports a large number of configuration options. These configurations options are controlled through parameters which are defined at design-time. A selection of the most important parameters of the simulation framework from the network interface perspective is provided in the table 5.1.

Parameter	**Description**
DIM_X	X Dimension of the architecture
DIM_Y	Y Dimension of the architecture
FLIT_SIZE	Flit size or link width (bits)
NoC_BUFFER_DEPTH	Number of flits which can be stored in a Router VC Buffer
NI_BUFFER_DEPTH	Number of flits which can be stored in a NI VC Buffer
VC	Number of virtual channels per port
TS	Number of time slots which define the scheduling cycle

Table 5.1.: Important parameters of the RTL simulation framework

Simulation framework is managed through tcl [110] and shell language scripts to ease the description of simulation scenario and underlying architecture. Modelsim simulation tool is used to elaborate the scripts, compile the sources and run the simulation [50]. Application code is loaded as a SRAM image file which can be accessed by the processor(s) in the tile. Collection and representation of simulation data are done in a semi-automatic manner to illustrate the results appropriately.

5.2. Hardware Prototyping

An FPGA prototype enables the verification of the proposed concepts. Real world applications are executed on the FPGA prototype for presented network interface architecture. In this work, we have used two FPGA prototyping solutions to validate our concepts. The details about the prototyping platforms are detailed in the following text.

5.2.1. Single FPGA prototype

Single FPGA prototype is used for investigations on relatively smaller architecture sizes. Xilinx ML605 development board was chosen for the single FPGA-based prototyping. This board contains a Xilinx Virtex-6 LX240T FPGA. The external interfaces like JTAG, UART and Ethernet are also available on the development board. The JTAG interface is used for programming and debugging on the FPGA board. This development board offers simpler and less time-consuming design flow for generating bitstreams of the underlying architecture as compared to multi-FPGA prototype, which will be described later. In addition, the debugging of the design on the single FPGA prototype is easier which makes the iteration cycle for the changes smaller.

The tiled architecture prototype is shown in the figure 5.1. The architecture consists of an incarnation of the Network on Chip. Each of the NoC routers connects to one of the tiles, developed for the InvasIC architecture. In our case, 4 tiles can be realized on the single FPGA prototype. Where each compute tile contains two LEON3 processing cores. One memory tile is present to establish the interface with DDR memory. The details about the InvasIC architecture and the subsequent components are already provided in the section 2.4.3. The realized network interface is used in each tile to connect the processing cores with the Network on Chip router.

The important parameters for the network interface and Network on Chip are chosen according to the settings shown by the table 5.2. The description of the above parameters is provided in table 5.1. The prototype uses the memory map, defined in the table 4.1.

Parameter	Value
DIM_X	2
DIM_Y	2
FLIT_SIZE	32
NoC_BUFFER_DEPTH	4
NI_BUFFER_DEPTH	512
VC	4
TS	8

Table 5.2.: FPGA prototype network interface and Network on Chip parameters

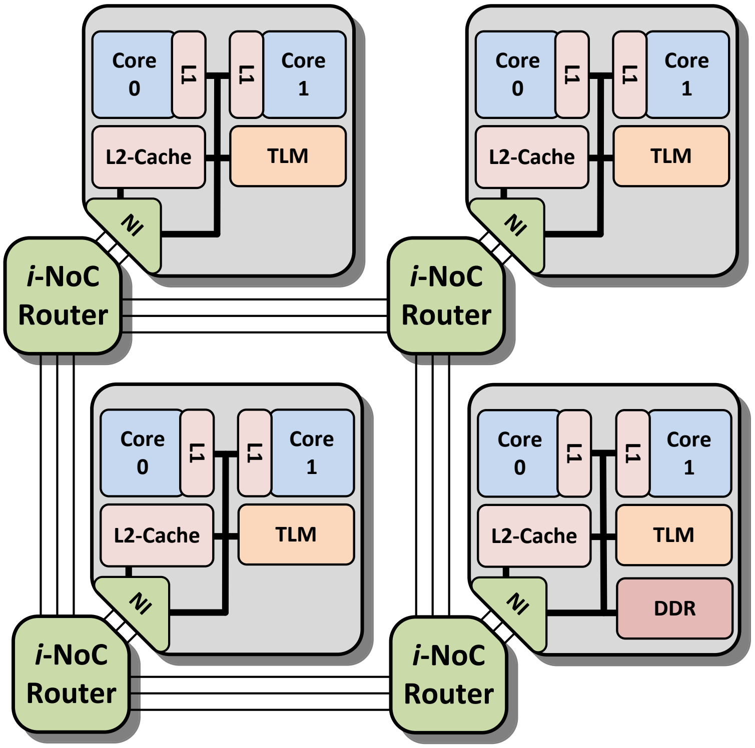


Figure 5.1.: Tiled architecture prototype on a single FPGA platform

Due to large resource utilization of the involved components in the architecture, the realizable tiled architecture size on single FPGA is relatively small. The main limitation for increasing the architecture size is the limited amount of memory which is available on the FPGA in the form of block RAMs. Block RAMs are used to realize the caches and the tile local memory in the architecture. Hence, the size of the caches and the tile local memories is limited in the prototyped architecture. On-chip storage in the form of tile local memories and caches is desired to keep the binaries and frequently accessed data. Limitation in the cache and TLM sizes reduces the performance which can be demonstrated on the prototyped architecture. In addition, the number of benchmarks, which can be executed on the underlying platform, becomes limited. The number of the physical interfaces which can be used for debugging on the development board are also limited. Table 5.3 gives an overview of the important configurations related to the caches and memories of the single FPGA prototype.

The clock frequency of the prototype is set to 50 MHz. The tile architecture with the configuration shown in the figure 5.1 consumes 39% of the registers available on the LX240T FPGA. Whereas, 71% of the FPGA lookup tables are required for the design. The limiting factor is the block RAM utilization, which is 89%.

Memory Type	Value
L1D Cache (Sets, Linesize, Lines)	2, 4, 4
L1I Cache (Sets, Linesize, Lines)	2, 8, 8
L2 Cache (Sets, Linesize, Lines)	2, 32, 8
Tile Local Memory	512kB
DDR3 Memory Size	1 GB

Table 5.3.: FPGA prototype memory configurations

5.2.2. CHIP-IT Prototype

As discussed in the section 5.2.1, the single FPGA prototype has limited capabilities with respect to the size of the architecture. Therefore, a scalable prototyping approach is required to enable prototyping and evaluation of many-core architectures [6], [41], [42]. A professional multi-FPGA prototyping solution manufactured by the company Synopsys, the CHIPit Platinum Edition, is used to prototype relatively larger tiled architecture [130]. In order to overcome the limitation of on-chip memory, SSRAM boards can be connected to the system. Global shared memory is available in the form of DDR memory. The prototyping platform consists of six Virtex-5 LX330 FPGAs as reconfigurable logic resources.

The prototyping solution provides standard I/O interfaces like DVI, Ethernet, and UART through extension boards. In addition, a custom interface in the form of Universal Multi-Resource (UMR) bus is present which offers scalable communication possibilities between the logic on the prototyping system and the software executed on the host computer. The UMR interface makes the debugging of the tiled architecture a lot convenient. Therefore, the CHIPit prototyping system can be used to realize a larger architecture. A mesh architecture of maximum size 3x2 could be implemented. The maximum size is selected keeping in view the six FPGAs and the symmetry of the tiled architecture. Currently, one tile is mapped on a single FPGA to uniformly access the SSRAM memory boards. Clock frequency is set to 25 MHz because of the DDR memory controller limitation. Figure 5.2 shows the multi-FPGA prototype on CHIPit platform.

5.3. Validation on RTL Simulation Framework

In this section, we have presented our investigations related to the software communication support on RTL simulation framework. Both case studies for software communication support i.e. remote data transfer for prefetching and remote task spawning are individually evaluated. In the RTL simulation framework for tiled architecture, the interface between software and hardware is modeled and simulated at finer abstraction level as compared to the SystemC-based framework which was discussed in section 3.2.3. AUTO_GS concept is independent of software interfaces and hence it is not considered for following investigations.

The tiled architecture modeled at RTL level including the proposed network interface architecture is implemented. The details about the simulation framework are already

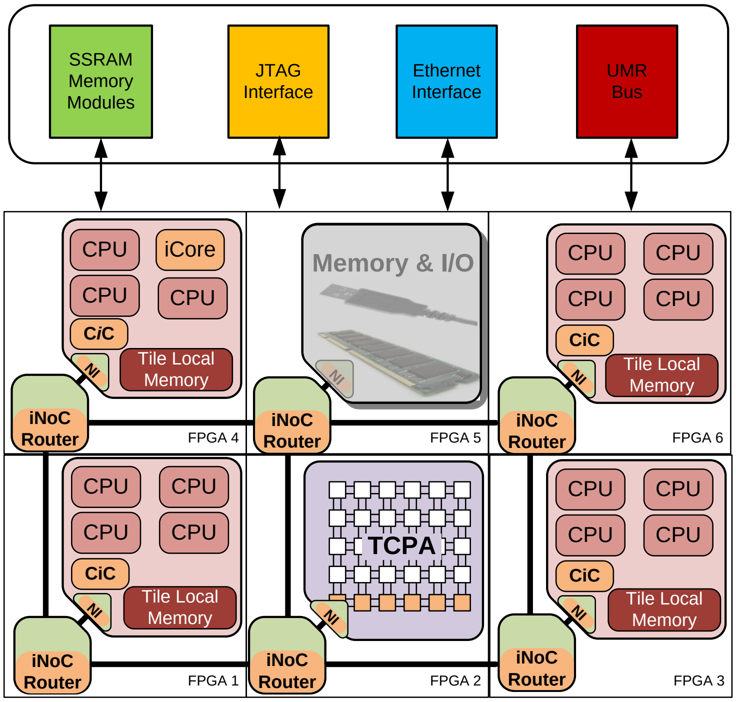


Figure 5.2.: Tiled architecture prototype on CHIPit platform

given in the section 5.1. iNoC RTL model with mesh topology and XY-routing is used for the following investigations. The size of the platform is configurable. We have applied uniform and hotspot traffic models which are common traffic benchmarks for the system evaluations where the NoC-based platforms are deployed [119] [23].

In the first step, hardware support for remote data transfer is evaluated. In the uniform scenario, each tile prefetches the data from every other tile in the architecture through direct memory access operation. The order in which the tiles perform the prefetch operation is defined. The amount of payload, which is prefetched is fixed. All tiles start at the same time. For the hotspot scenario, all tiles transfer data to the same destination tile. The corner tile (0, 0) is chosen as the common destination tile. The simulation is stopped after all tiles are finished. We have compared our concept with the state of the art approach proposed by Varghese et al. [134]. The methodology used by Varghese et al. is named as Reference (Ref). In the Reference configuration, the synchronization related to the data transfer is managed by the software. Our approach in which the signaling for remote data transfer is handled by the network interface is named as NIPF. In both configurations, data transfer is done through the direct memory access hardware engine.

In the second step, the proposed task spawning hardware support is investigated. In the uniform case, each tile spawns a given number of tasks on every other tile in the architecture in a defined sequence. Each spawned task corresponds to a defined computational workload. Task spawning consists of communication between the source and destination tile in the form of three steps as mentioned in the section 3.2.2. All tiles start communicating at the same time. trig_task_pointer_src operation indicates the completion of a task spawning request. For the hotspot scenario, all tiles attempt to spawn tasks on the same destination tile. The corner tile (0, 0) is chosen as the hotspot. The simulation is stopped after all tiles are finished. We have compared our approach with state of the art approach proposed by Kavadias et al. [78]. The approach presented by Kavadias et al. is named as Reference (Ref). In state of the art approach, the synchronization related to task spawning is handled by the software. Our approach in which the task spawning is done by the hardware support in the network interface is named as NITS. Both approaches use RDMA hardware engine to move task data between tiles. The payload which is transferred as task data during task spawning is kept fixed.

5.3.1. Discussion of Results

Figures 5.3 and 5.4 show the comparison of the execution time for remote data transfers between the two configuration for uniform and hotspot traffic respectively. The architecture size is increased in different iterations to get further insight on the performance of the proposed hardware support.

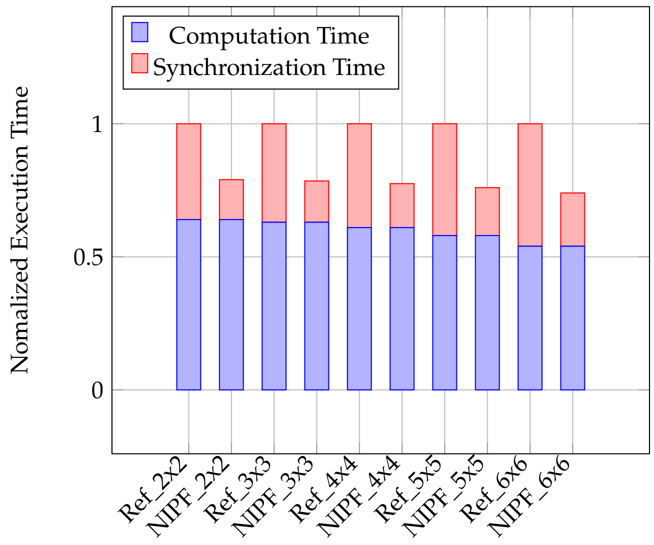


Figure 5.3.: Execution time of uniform scenario for data prefetching support evaluations

Represented execution time is normalized with respect to the execution time in Reference configuration. The execution time of the benchmarks is further categorized in the computation time and the synchronization time. The time which is spent on the processing

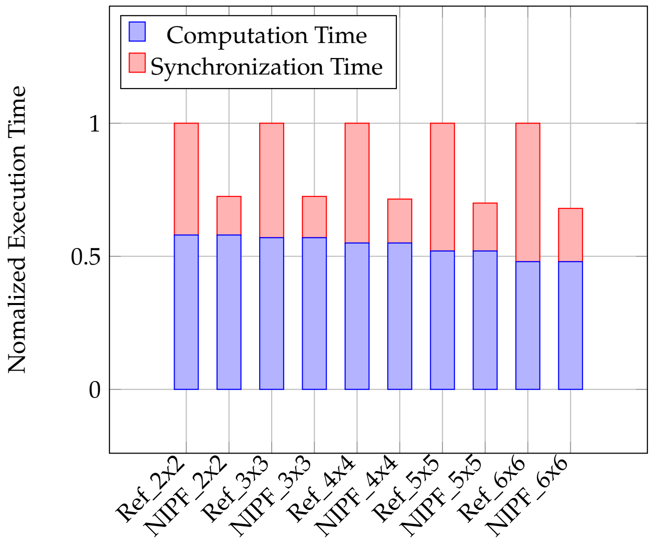


Figure 5.4.: Execution time of hotspot scenario for data prefetching support evaluations

of the task workload and queuing delays in the architecture is indicated as computation time. The latency which results from the synchronization overhead for data transfer operation between remote tiles is marked as synchronization time. In the uniform scenario, the increase in synchronization time is linear with the increasing architecture size. Our approach gives the maximum performance gain of up to 26% as compared to Reference in the uniform scenario. The results for hotspot scenario show a higher increase in synchronization time when the architecture size is increased. Our approach gives a performance improvement of up to 32% in comparison to Reference. The improvement comes from the fact that the software is offloaded from managing the status of data transfer completion. The hardware support introduced inside the network interface handles the bookkeeping of the data transfer status and indicates the completion of data transfer through an interrupt to the software. The simulation scenarios which have a higher degree of parallelism, i.e., more data transfer operations are performed between remote tiles in a bigger architecture configuration, the synchronization time makes the larger share of the overall execution time in Reference configuration. In our approach, the tile interconnect is not loaded with status polling requests from software which results in the corresponding performance improvement.

Figures 5.5 and 5.6 show the investigations related to task spawning support for both the uniform and hotspot scenarios. The execution time of the Reference and the proposed approach is compared with increasing architecture size.

Similar to the evaluations related to remote data transfer, the execution time is classified in computation time and the synchronization time. The computation time refers to the delay which is agnostic from the proposed hardware extensions. The latency which comes

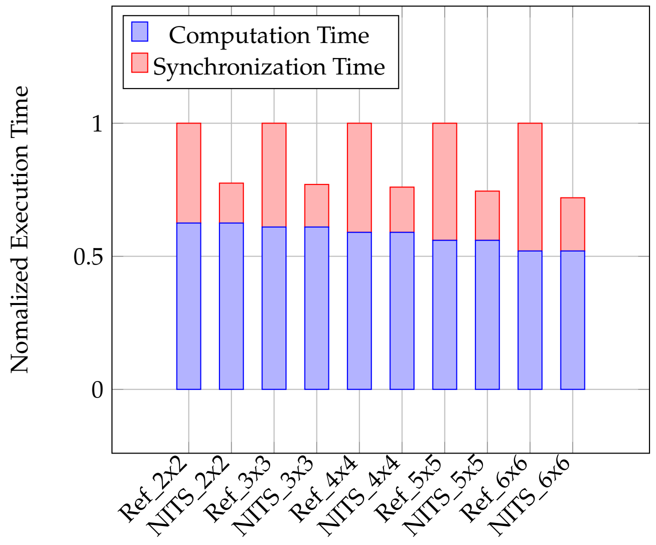


Figure 5.5.: Execution time of uniform scenario for task spawning support evaluations

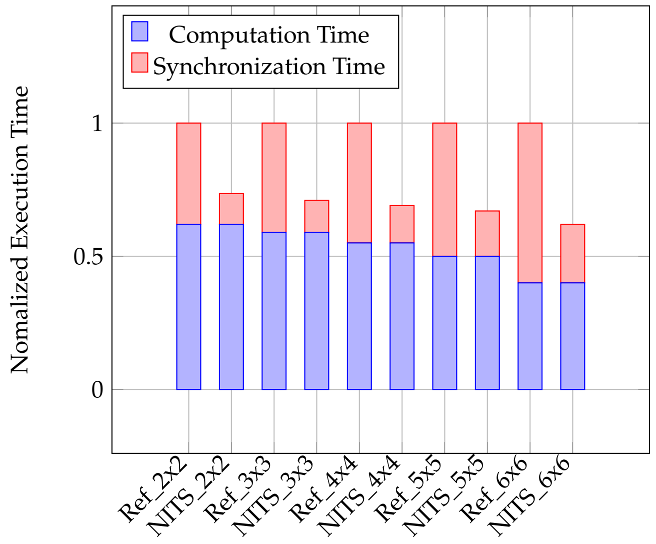


Figure 5.6.: Execution time of hotspot scenario for task spawning support evaluations

from the synchronization time during remote task spawning operation is called synchronization delay. The results for hotspot scenario show a higher increase in synchronization time with the increasing architecture size. In the uniform scenario, an improvement of up to 28% in execution time is observed. Our approach gives a performance gain of up to 38% in comparison to Reference for hotspot scenario. The simulation scenarios which have a higher degree of parallelism, i.e., more number of tasks are spawned to remote tiles in a bigger architecture configuration, the status polling requests make a significant share of traffic on tile interconnect and hence result in higher synchronization time in the Reference configuration. Our approach offloads the software from checking the status of task spawning sub-operations. Hence, the tile interconnect is not loaded with status polling requests which results in the corresponding performance improvement in our configuration.

5.4. Validation on FPGA Prototype

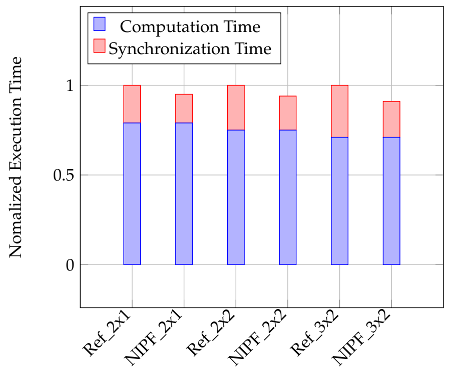
In addition to the simulation framework, we investigated the proposed hardware extensions for software communication support on the FPGA prototype. A parallel implementation of an integer matrix multiplication application is executed on the target platform to demonstrate the benefits of proposed concepts. For the following investigations, the matrix size is varied in three different configurations. Larger matrix size scenarios generate higher workload on the underlying system as compared to the scenarios with smaller matrix size. Multi-FPGA prototyping system as described in section 5.2.2 is used for prototyping. A 1 GB DDR memory is used as global shared memory which is present as a memory tile among other compute tiles in the architecture. The matrices and the code of the parallel application are loaded in the global shared memory before the execution. Each compute tile consists of 4 LEON3 Sparc V8 cores [43]. In addition, each compute tile contains an 8 MB tile local memory in which the corresponding task data and code are copied from global shared memory during execution. iNoC is used as the Network on Chip for exchanging data between different tiles. The network interface is used to connect the tiles with the Network on Chip.

For data transfer evaluations, the application code and matrices are copied from the memory tile to the respective tile local memories through remote direct memory access hardware accelerator. Task pointers pointing to the address of application code in tile local memory are statically assigned. After the execution is completed, each tile writes its results back to the memory tile. For task spawning, the memory tile spawns tasks of matrix multiplication application to each tile in the architecture. During task spawning, the application code and the task data i.e. matrices are copied from the memory tile to the tile local memories through RDMA unit. However, instead of using static values, task pointers carrying the start address of the application code in the tile local memory are sent via Task i-lets. Finally, each tile writes the computation results back to the memory tile.

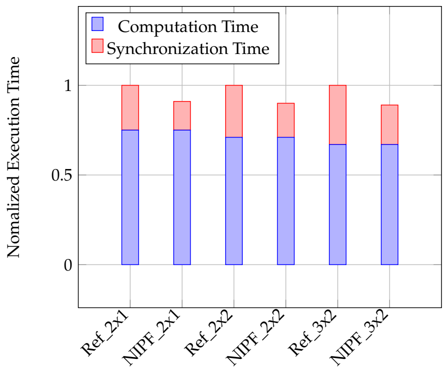
5.4.1. Discussion of Results

Figure 5.7 shows the execution time of matrix multiplication for different architecture sizes corresponding to asynchronous data transfer scenario. As compared to the simulation framework, only certain configurations of architecture sizes can be realized on the

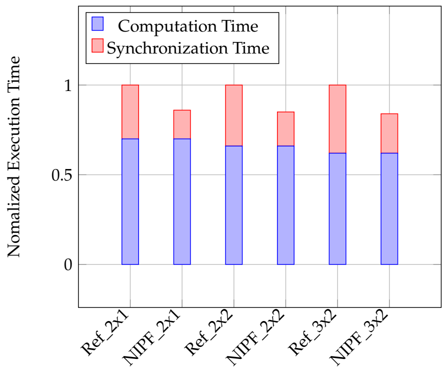
FPGA platform because of prototyping area limitation. Ref and NIPF refer to the state of the art and the proposed configurations respectively. Both configurations are individually executed in different architecture size variants on the FPGA prototype. Execution time is normalized with respect to the corresponding Reference configuration. The computation time represents the time taken by the software on the processing cores for matrix multiplication. Computation time is the same in both compared configurations. The synchronization time represents the delay for managing the status of data transfer operation.



(a) Matrix_size = 32x32



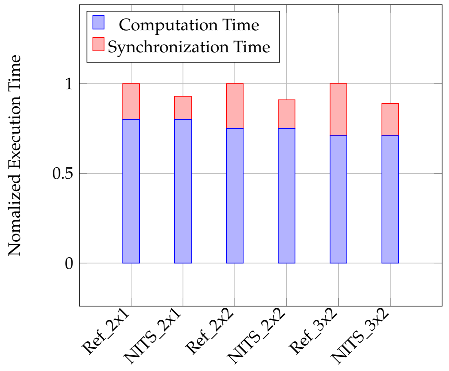
(b) Matrix_size = 64x64



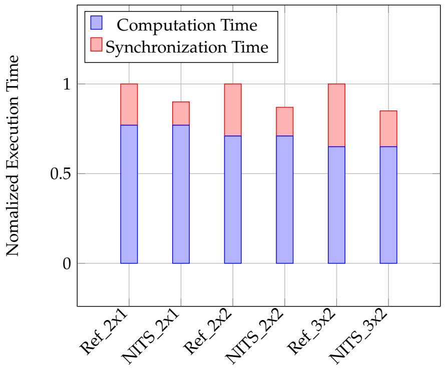
(c) Matrix_size = 128x128

Figure 5.7.: Execution time of matrix multiplication application for data prefetching support evaluations

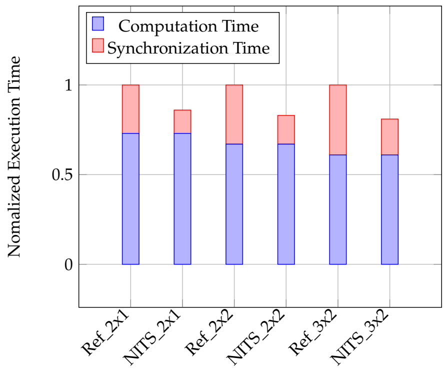
Our approach gives a speedup of up to 16% as compared to the Reference. The speedup comes from the fact that the matrices can be prefetched from the global shared memory to the tile local memory without blocking the software execution. The introduced hardware support offloads the software from signaling operations required during remote data transfer. The synchronization overhead makes a larger share of overall execution time in case of bigger matrix sizes. Therefore, it can be observed that the scenarios with larger matrix size show more improvement as compared to the ones with smaller matrix size.



(a) Matrix_size = 32x32



(b) Matrix_size = 64x64



(c) Matrix_size = 128x128

Figure 5.8.: Execution time of matrix multiplication application for task spawning support evaluations

Figure 5.8 shows the execution time of matrix multiplication application for the task spawning case study. Ref and NITS refer to the state of the art and the proposed configurations respectively. Our methodology reduces the synchronization time and thus delivers a performance improvement of up to 19% as compared to the Reference configuration. The synchronization time is reduced by using efficient handshaking and signaling mechanisms by our concept. In the Reference configuration, software supervises the entire synchronization required for different phases of task spawning. In our methodology, the bookkeeping of each sub-operation is handled by the specific hardware support. Therefore, the software is offloaded from synchronizing events between remote tiles and can proceed with the matrix multiplication processing. In addition, it can be observed that our approach becomes more beneficial in scenarios with higher workload i.e. larger matrix sizes. The reason being larger synchronization overhead in these scenarios as compared to the scenarios with smaller matrix sizes.

Looking at the results, it can be observed that the achieved speedup resulting from our contributions increases as we move to bigger architecture configurations and larger workload scenarios. These scenarios exploit the task level parallelism at a finer granularity. Therefore, it can be stated that our concepts provide better scalability as compared to the state of the art approaches when the degree of parallelism increases.

6. Conclusion and Outlook

6.1. Conclusion

Distributed shared memory architectures have revolutionized the way in which modern many-core systems could be programmed. They make it possible for the application developers to play around with both legacy shared memory and message passing programming models and tune their applications on the underlying architecture. This helps in reducing Time-to-Market and hence increases the overall productivity of the development cycle. However, the introduction of distributed shared memory approach comes up with two important challenges; 1) management of resources in underlying communication infrastructure and 2) synchronization overhead between software instances running on remote tiles. In this work, we come up with an approach in which architectural support is provided in the network interface to tackle the above-mentioned challenges. In the following sections, we conclude our findings and provide a future outlook of this work.

6.2. Management of communication resources

Network on Chip contains virtual channels as communication resources which are shared between concurrent traffic flows and can be used to increase network throughput. Efficient management of communication resources is one of the most important challenges for Network on Chip based architectures. Network on Chip management points to the assignment of virtual channels between guaranteed service and best effort communication. State of the art methods allocate communication resources without considering the impact of their decisions on the utilization of underlying communication infrastructure. In distributed shared memory systems, communication between application instances results in traffic patterns which are difficult to predict before execution. Therefore, state of the art approaches for communication resource management result in inefficient virtual channel utilization and hence reduced overall performance.

We have proposed a concept which relies on self-optimization principles to assign virtual channels between concurrent communication flows. Our approach regards dynamically changing communication patterns while performing communication resource allocation. Traffic is monitored at run-time and end-to-end connections are established on the basis of monitoring data to optimize the usage of communication infrastructure. Connections based on current traffic scenario are named as AUTO_GS connections. Proposed concept is implemented in the form of hardware extensions in the network interface architecture. The effectiveness of the approach is demonstrated with experiments using synthetic traffic and real world applications. Our evaluations highlight that the assignment of virtual channels through our concept leads to better results in terms of communication infrastructure utilization, latency, and power consumption. Proposed hardware extensions are prototyped for FPGA and ASIC platforms. RTL implementation results depict that the hardware modules corresponding to our concept are synthesizable with reasonable area cost at the target platforms without significant reduction in the maximum achievable frequency.

6.3. Synchronization support for remote software instances

With increasing task level parallelism, the synchronization overhead between software instances running on different tiles becomes significant in defining overall system performance. Therefore, software synchronization support in single chip distributed shared memory architectures is the second contribution of this work. In particular, the synchronization overhead between software layers during remote data transfer and remote task spawning operations has been focused. State of the art approaches for synchronization support require software involvement at every stage of the remote data transfer operation, which results in performance degradation. To the best of our knowledge, the performance overhead associated to synchronize software instances during remote task spawning operation over DSM architectures has not been addressed by any state of the art approach.

We have proposed the strategy to offload software from handshaking operations, which are required during data transfer between remote tiles. The hardware support inside network interface architecture has been extended to supervise the status of remote data transfers. Secondly, we have introduced hardware extensions for managing synchronization activities, which are required at multiple stages of the remote task spawning operation. Simulation framework, as well as the FPGA prototype, are used for concept evaluation. Real world applications are executed on the target platforms to validate the usefulness of our concept. Results show that the proposed hardware support leads to lesser execution time and reduced energy consumption on the underlying architecture when compared with the state of the art approaches. Synthesis results highlight the low footprint requirements of the proposed hardware modules on FPGA and ASIC targets.

6.4. Future work

The proposed network interface architecture with its novel ingredients is being used in the Trans-regional Collaborative Research Center 89 'Invasive Computing' [131]. The future work which is described in the following will be addressed in the second phase of this research project.

6.4.1. Configurable cache coherence support

Shared memory model continues to be the dominant programming paradigm in the modern many-core architectures. The fact that the memory is physically distributed makes the cache coherence an important challenge for deploying shared memory programming model on distributed shared memory systems. State of the art methods with software-based cache coherence lead to higher performance overhead in distributed shared memory architectures. Therefore, cache coherence hardware support is essential to benefit from shared memory programming. Conventional methods for supporting system-wide cache coherence result in poor performance due to the overhead of cache coherence protocol over distributed communication medium. Hence, it becomes vital to devise intelligent strategies which ensure better performance by reducing the overhead of cache coherence support over Network on Chip. In recent times, researchers have focused on developing new methodologies to support cache coherence on distributed on-chip systems [28] [30] [24]. As a future work, it is planned to explore novel strategies for cache coherence hardware support for distributed shared memory platforms.

A concept is under investigation which enables the management of cache organization in an application-aware manner. According to this methodology, cache coherence support within the architecture would be configured through high level "Hints" from the operating system. This approach has been named as *region-based cache coherence*. Region-based cache coherence concept configures the hardware support for cache coherence selectively in the architecture nodes, where it is desired. The group of nodes where the cache coherence is configured is named as a *region*. This concept provisions that the coherence traffic can be handled at a high priority level only within the region. This aspect helps in reducing the overhead of cache coherence support in comparison to the schemes which advocate system-wide cache coherence. Region-based cache coherence concept is shown in the figure 6.1. In addition, the idea of considering the utilization of cached data while providing cache coherence support is being considered [29]. The cache hierarchy, as well as the communication infrastructure, will be extended to support the configurable cache coherence. In particular, the focus of our future work will be on the hardware extensions, which are required in the network interface architecture.

6.4.2. Synchronization support during task execution

To exploit available parallelism over the given platform, an application spawns multiple tasks which are mapped on the available cores either within the same compute tile or in different compute tiles. Tasks which are mapped in different tiles result in higher execution time because they need to communicate over Network on Chip. In the current work, we have presented concepts to reduce the synchronization overhead during data transfer and task spawning between remote tiles. Besides the above-mentioned phases, remote tasks also require synchronization support during other phases of their execution. The access to a shared object is the common example where the remote tasks need to synchronize. In situations, where underlying platform lacks the appropriate synchronization support, the overall performance gain from task level parallelism reduces because of the synchronization overhead. Therefore, efficient mechanisms are needed which address inter-tile synchronization during task execution [95].

Software based synchronization methods are static and do not consider the characteristics of underlying architecture. Therefore, these methods can not harness fine-grained parallelism. In addition, software-based synchronization leads to degraded performance

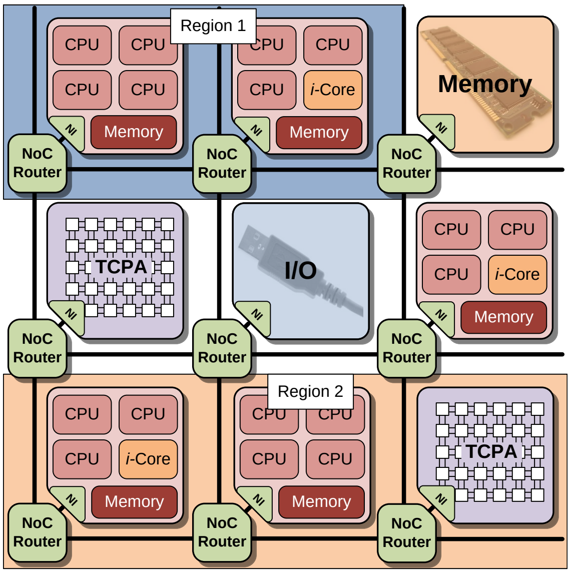


Figure 6.1.: Region-based cache coherence

over Network on Chip based architectures. This is due to the fact that the actual application execution is suspended the majority of the time because of communication delays for exchanging messages over NoC. Hence, hardware-assisted synchronization mechanisms are desired which offload the software from supervising operations which result in inter-tile communication [148]. Hardware extensions can be introduced which enable conventional synchronization mechanisms like MUTEX, test-and-set or compare-and-swap between the tiles and result in lesser synchronization overhead as compared to pure software based implementation [143]. Network interface architecture can be extended to support the exchange of light weight synchronization messages between remote tiles. In addition, dedicated hardware support can be developed in the network interface to perform lock operations and hence offloading the software from executing costly lock sub-routines [147].

A. Appendix

A.1. AHB Bus Signals

Important AHB signals with their description is provided in the table A.1

Signal Name	Source	Description
HADDR[31:0]	Master	32 bit address bus
HWDATA[31:0]	Master	Data bus for write operations, can have size from 32 to 1024 bits
HTRANS[1:0]	Master	Type of the current transfer, can be Non-Sequential, Sequential, Idle or Busy
HSIZE[2:0]	Master	Size of the current transfer, transfers are supported from size of 8 bits to 1024 bits
HBURST[2:0]	Master	Type of the burst transfer, four, eight and sixteen beat burst transfers are supported
HBUSREQ	Master	Indication of master to the arbiter that it wants an access of the bus
HRDATA[31:0]	Slave	Data bus for read operations, can have size from 32 to 1024 bits
HREADY	Slave	High status of this signal indicates completion of the transfer, can be held LOW to extend the transfer
HRESP[1:0]	Slave	Status of the transfer, can be Okay, Error, Retry or Split
HSPLIT	Slave	used to indicate that which master should repeat the split transaction
HGRANT	Arbiter	This signal indicates that which master is granted access to the bus
HMASTER[3:0]	Arbiter	This signal from arbiter is used by the bus slaves which support split transfer to determine that which master is performing the transfer
HSEL	Decoder	Status of the transfer, can be Okay, Error, Retry or Split

Table A.1.: Important AHB signals [126]

A.2. RTL Code Hierarchy

RTL Code hierarchy of the network interface is presented in Figure A.1.

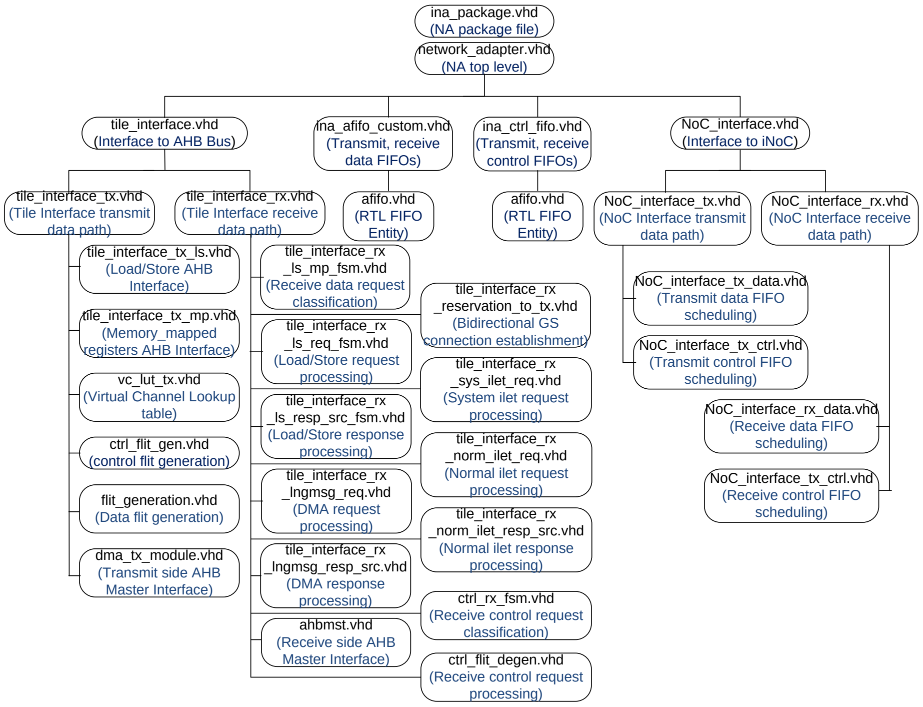

ina_package.vhd (NA package file)
network_adapter.vhd (NA top level)
tile_interface.vhd (Interface to AHB Bus)
ina_afifo_custom.vhd (Transmit, receive data FIFOs)
ina_ctrl_fifo.vhd (Transmit, receive control FIFOs)
NoC_interface.vhd (Interface to iNoC)
tile_interface_tx.vhd (Tile Interface transmit data path)
tile_interface_rx.vhd (Tile Interface receive data path)
afifo.vhd (RTL FIFO Entity)
afifo.vhd (RTL FIFO Entity)
NoC_interface_tx.vhd (NoC Interface transmit data path)
NoC_interface_rx.vhd (NoC Interface receive data path)
tile_interface_tx_ls.vhd (Load/Store AHB Interface)
tile_interface_tx_mp.vhd (Memory_mapped registers AHB Interface)
vc_lut_tx.vhd (Virtual Channel Lookup table)
ctrl_flit_gen.vhd (control flit generation)
flit_generation.vhd (Data flit generation)
dma_tx_module.vhd (Transmit side AHB Master Interface)
tile_interface_rx _ls_mp_fsm.vhd (Receive data request classification)
tile_interface_rx _ls_req_fsm.vhd (Load/Store request processing)
tile_interface_rx _ls_resp_src_fsm.vhd (Load/Store response processing)
tile_interface_rx _lngmsg_req.vhd (DMA request processing)
tile_interface_rx _lngmsg_resp_src.vhd (DMA response processing)
ahbmst.vhd (Receive side AHB Master Interface)
tile_interface_rx _reservation_to_tx.vhd (Bidirectional GS connection establishment)
tile_interface_rx _sys_ilet_req.vhd (System ilet request processing)
tile_interface_rx _norm_ilet_req.vhd (Normal ilet request processing)
tile_interface_rx _norm_ilet_resp_src.vhd (Normal ilet response processing)
ctrl_rx_fsm.vhd (Receive control request classification)
ctrl_flit_degen.vhd (Receive control request processing)
NoC_interface_tx_data.vhd (Transmit data FIFO scheduling)
NoC_interface_tx_ctrl.vhd (Transmit control FIFO scheduling)
NoC_interface_rx_data.vhd (Receive data FIFO scheduling)
NoC_interface_tx_ctrl.vhd (Receive control FIFO scheduling)

Figure A.1.: Network interface code hierarchy

A.3. Network Interface Driver Library

```
#include <stdio.h>
#include <stdlib.h>

#define SRC_ADDR 0x80010000
#define DST_LOCAL_ADDR 0x80010000
#define DST_GLOBAL_ADDR 0x41010000
#define DST_GLOBAL_ADDR1 0x40010000
#define DDR_ADDR 0x00000000
#define DATA_SIZE 0x20

 volatile unsigned int *tile_id = (volatile unsigned int *) 0x80E00000;

/*    GS Reservation Registers */

  volatile unsigned int *Dst_naddr_reg = (volatile unsigned int *) 0x80E00108;
  volatile unsigned int *Sl_reg = (volatile unsigned int *) 0x80E0010C;

 /* registers for reading connection status, based on destination id */
 volatile unsigned int *VC_DST_REG = (volatile unsigned int *) 0x80E00400;
 volatile unsigned int *VC_DST_STATUS_REG = (volatile unsigned int *) 0x80E00404;

 /* registers for reading connection status, vc based */

volatile unsigned int *VC0_STATUS_REG = (volatile unsigned int *) 0x80E00500;
volatile unsigned int *VC1_STATUS_REG = (volatile unsigned int *) 0x80E00504;
volatile unsigned int *VC2_STATUS_REG = (volatile unsigned int *) 0x80E00508;
volatile unsigned int *VC3_STATUS_REG = (volatile unsigned int *) 0x80E0050C;

/*    Long Msg Registers */

  volatile unsigned int *LngMsg_msg_id_reg = (volatile unsigned int *) 0x80E00300
      ;
  volatile unsigned int *LngMsg_Payload_length = (volatile unsigned int *) 0
      x80E00304;
  volatile unsigned int *LngMsg_Payload_addr   = (volatile unsigned int *) 0
      x80E00308;
  volatile unsigned int *LngMsg_Payload_Dst_addr = (volatile unsigned int *) 0
      x80E0030C;
  volatile unsigned int *LngMsg_status_reg       = (volatile unsigned int *) 0
      x80E00320;
  volatile unsigned int *LngMsg_status_recv_reg  = (volatile unsigned int *) 0
      x80E00324;

 /*    Normal i-let destination Indirect Registers */

  volatile unsigned int *norm_ilet_dst_indir_msg_id_reg  = (volatile unsigned int
       *) 0x80E00600;
  volatile unsigned int *norm_ilet_dst_indir_dst_id_reg = (volatile unsigned int
      *) 0x80E00604;
  volatile unsigned int *norm_ilet_dst_indir_payload_reg1 = (volatile unsigned
       int *) 0x80E00610;
  volatile unsigned int *norm_ilet_dst_indir_payload_reg2 = (volatile unsigned
       int *) 0x80E00614;
  volatile unsigned int *norm_ilet_dst_indir_payload_reg3 = (volatile unsigned
       int *) 0x80E00618;
  volatile unsigned int *norm_ilet_dst_indir_payload_reg4 = (volatile unsigned
       int *) 0x80E0061C;
  volatile unsigned int *norm_ilet_dst_indir_status_reg = (volatile unsigned int
       *) 0x80E00620;

 /*    Normal i-let source Indirect Registers */

  volatile unsigned int *norm_ilet_src_indir_msg_id_reg  = (volatile unsigned int
        *) 0x80E00700;
  volatile unsigned int *norm_ilet_src_indir_payload_reg1 = (volatile unsigned
       int *) 0x80E00710;
  volatile unsigned int *norm_ilet_src_indir_payload_reg2 = (volatile unsigned
       int *) 0x80E00714;
  volatile unsigned int *norm_ilet_src_indir_payload_reg3 = (volatile unsigned
       int *) 0x80E00718;
  volatile unsigned int *norm_ilet_src_indir_payload_reg4 = (volatile unsigned
       int *) 0x80E0071C;
  volatile unsigned int *norm_ilet_src_indir_status_reg = (volatile unsigned int
       *) 0x80E00720;

/*    Normal i-let direct Registers */

  volatile unsigned int *norm_ilet_dir_msg_id_reg  = (volatile unsigned int *) 0
       x80E00800;
  volatile unsigned int *norm_ilet_dir_dst_id_reg = (volatile unsigned int *) 0
       x80E00804;
  volatile unsigned int *norm_ilet_dir_payload_reg1 = (volatile unsigned int *) 0
       x80E00810;
  volatile unsigned int *norm_ilet_dir_payload_reg2 = (volatile unsigned int *) 0
       x80E00814;
  volatile unsigned int *norm_ilet_dir_payload_reg3 = (volatile unsigned int *) 0
       x80E00818;
  volatile unsigned int *norm_ilet_dir_payload_reg4 = (volatile unsigned int *) 0
       x80E0081C;
  volatile unsigned int *norm_ilet_dir_status_reg = (volatile unsigned int *) 0
       x80E00820;

/*    System i-let Registers */

  volatile unsigned int *sys_ilet_dst_id_reg = (volatile unsigned int *) 0
       x80E00904;
  volatile unsigned int *sys_ilet_payload_reg1 = (volatile unsigned int *) 0
       x80E00910;
  volatile unsigned int *sys_ilet_payload_reg2 = (volatile unsigned int *) 0
       x80E00914;
  volatile unsigned int *sys_ilet_payload_reg3 = (volatile unsigned int *) 0
       x80E00918;
  volatile unsigned int *sys_ilet_payload_reg4 = (volatile unsigned int *) 0
       x80E0091C;
  volatile unsigned int *sys_ilet_payload_reg5 = (volatile unsigned int *) 0
       x80E00920;
  volatile unsigned int *sys_ilet_payload_reg6 = (volatile unsigned int *) 0
       x80E00924;
  volatile unsigned int *sys_ilet_payload_reg7 = (volatile unsigned int *) 0
       x80E00928;
  volatile unsigned int *sys_ilet_payload_reg8 = (volatile unsigned int *) 0
       x80E0092C;
  volatile unsigned int *sys_ilet_src_status_reg = (volatile unsigned int *) 0
       x80E00930;
  volatile unsigned int *sys_ilet_dst_status_reg = (volatile unsigned int *) 0
       x80E00940;
  volatile unsigned int *sys_ilet_dst_ptr_reg = (volatile unsigned int *) 0
       x80E00944;

main() {

  if ((*tile_id == 0)) {
    printf("TILE%d: Hello!\n", *tile_id);

    volatile unsigned int* store_addr;
    volatile unsigned int* ddr_addr;
    unsigned int idx,iter;
    unsigned int i,j;

/********     Writing known pattern at Local DMA Address for DMA Testing ******
     */
    printf("Store at DMA Address first ...\n");
    for (idx=0; idx<DATA_SIZE;idx++){
    store_addr = (unsigned int*)(SRC_ADDR+idx*4);
    *store_addr = (DST_LOCAL_ADDR+idx*4);
    }
  printf("Store complete ...\n");

/***************** Connection Reservation test ************************/

    *Dst_naddr_reg = DST_GLOBAL_ADDR; // type casting required does not work
         otherwise
    *Sl_reg = 0x00050001; // SL = 1, invade = 1, bidirectional = 0
     printf("Initiated Connection Reservation\n");


/************************ Read Connection Status *********************/
      *VC_DST_REG = DST_GLOBAL_ADDR;
      printf("Connection Status is %x \n",*VC_DST_STATUS_REG & 0xE0000000);

/*Connection status may be checked by checking the status of VCs individually */
   printf("Connection Status is %x \n",*VC0_STATUS_REG & 0xE0000000);


/************* Initiating DMA over reserved connection **************/
```

```
      while(*LngMsg_status_reg !=0);
      *LngMsg_msg_id_reg = 0x0;
     *LngMsg_Payload_length = DATA_SIZE;
     *LngMsg_Payload_addr  = SRC_ADDR;
     *LngMsg_Payload_Dst_addr = DST_GLOBAL_ADDR;
      while(*LngMsg_status_recv_reg !=0);
      printf("DMA Request Initiated to Tile 1\n");


// Connection release test

     *Dst_naddr_reg = DST_GLOBAL_ADDR;
     *Sl_reg = 0x00040001; // SL = 1, invade = 0 (retreat), bidirectional = 0
      printf("Initiated Connection Release\n");

/****************  Read Connection status after releasing *****************/
       *VC_DST_REG = DST_GLOBAL_ADDR;
       printf("Connection Status is %x \n", *VC_DST_STATUS_REG & 0xE0000000);

    printf("Connection Status is %x \n",*VC0_STATUS_REG & 0xE0000000);
    printf("Connection Status is %x \n",*VC1_STATUS_REG & 0xE0000000);
    printf("Connection Status is %x \n",*VC2_STATUS_REG & 0xE0000000);
    printf("Connection Status is %x \n",*VC3_STATUS_REG & 0xE0000000);


/****************** Sending System i-let with dummy payload **************/

     while(*sys_ilet_src_status_reg !=0);
     printf("System i-let Initiated to Tile 1\n");
     *sys_ilet_dst_id_reg = 0x41000000;
     *sys_ilet_payload_reg1 =  0x1;
     *sys_ilet_payload_reg2 =  0x43;
     *sys_ilet_payload_reg3 =  0x44;
     *sys_ilet_payload_reg4 =  0x45;
     *sys_ilet_payload_reg5 =  0x46;
     *sys_ilet_payload_reg6 =  0x47;
     *sys_ilet_payload_reg7 =  0x48;
     *sys_ilet_payload_reg8 =  0x49;

/*Normal i-let which is following DMA tranfer before, Task spawning support */
     printf("i-let Enqueue Destination Indirect Request Initiated to Tile 1\n");
     *norm_ilet_dst_indir_msg_id_reg  = 0x0;
     *norm_ilet_dst_indir_dst_id_reg = 0x41800000;
     *norm_ilet_dst_indir_payload_reg1 = 0x1a;
     *norm_ilet_dst_indir_payload_reg2 = 0x2b;
     *norm_ilet_dst_indir_payload_reg3 = 0x3c;
     *norm_ilet_dst_indir_payload_reg4 = 0x4d;


/**************************** Normal i-let source Indirect *****/
     printf("i-let Enqueue Source Indirect Request Initiated to Tile 1\n");
     *norm_ilet_src_indir_msg_id_reg  = 0x0;
     *norm_ilet_src_indir_payload_reg1 = 0x5e;
    *norm_ilet_src_indir_payload_reg2 = 0x6f;
    *norm_ilet_src_indir_payload_reg3 = 0x7A;
    *norm_ilet_src_indir_payload_reg4 = 0x8B;

/************************ Normal i-let Destination Indirect *********/
    while(*norm_ilet_dir_status_reg !=0);
    printf("i-let Enqueue (direct) Initiated to Tile 1\n");
    *norm_ilet_dir_msg_id_reg  = 0x2;  // Msg-id for indirect i-lets can be
         different
    *norm_ilet_dir_dst_id_reg = 0x41800000;
    *norm_ilet_dir_payload_reg1 = 0x9C;
    *norm_ilet_dir_payload_reg2 = 0x10D;
    *norm_ilet_dir_payload_reg3 = 0x11E;
    *norm_ilet_dir_payload_reg4 = 0x12F;

}


if ((*tile_id == 1)) {
    printf("TILE%d: Hello!\n", *tile_id);
    volatile unsigned int* ddr_addr;
    unsigned int i,j;
    for (j=0; j<1000000; j++); // known delay
/****************** Verifying DMA written data **************************/
    for (i=0;i<DATA_SIZE;i++){
        ddr_addr = (unsigned int*)(DST_LOCAL_ADDR+i*4);
          if (*ddr_addr == ddr_addr )
            j = 1;
          else{
          printf("Error at Address 0x%x\t,0x%x\n",ddr_addr, *ddr_addr);
          j = 0;
           }
}
      if (j==1)
          printf("Test Successful\n");

  }

if ((*tile_id == 2)) {

    printf("TILE%d: Hello!\n", *tile_id);
}

if ((*tile_id == 3)) {
    printf("TILE%d: Hello!\n", *tile_id);
}

  return 0;

}
```

List of Figures

List of Tables

Abbreviations

AHB	advanced high-performance bus
AMBA	advanced microcontroller bus architecture
APB	advanced peripheral bus
ARM	advanced RISC machines
ASB	advanced system bus
ASIC	application-specific integrated circuit
ASP	advance peripheral bus

BE	best-effort
BU	buffer utilization

CiC	core ilet controllers
CISC	complex instruction set computer
CMOS	complementary metal oxide semiconductor
CMP	chip multiprocessor
CPU	central processing unit
CRE	communication related energy
CS	circuit switching

DDR	double data rate
DEMUX	demultiplexer
DFT	discrete Fourier transform
DMA	direct memory access
DSE	design space exploration
DSM	distributed shared memory
DSU	debug support unit
DVFS	dynamic voltage and frequency scaling
DVI	digital visual interface

ECC	error-correcting code
EPIC	explicitly parallel instruction computing
EVC	express virtual channel
FEC	forward error correction
FI	frequency island
FIFO	first in - first out
flit	flow control digit
FPGA	field programmable gate array
FPU	floating-point unit
FSM	finite state machine
GPU	graphics processing unit
GS	guaranteed service
GT	guaranteed throughput
HDL	hardware description language
HLS	high-level synthesis
HPC	high-performance computing
HRE	heterogeneous reconfigurable engine
i-NoC	invasive network on chip
i-NI	invasive network interface
I/O	input/output
IDN	input/output dynamic network
IET	independent execution time
IP	intellectual property
IRA	input reservation arbitration
iRTSS	invasive run-time support system
ISA	instruction set architecture
ITRS	International Technology Roadmap for Semiconductors
JTAG	joint test action group
LAN	local area network
LU	link utilization

LUT look up table

MC memory controller
MDN memory dynamic network
MPB message passing buffer
MPI message passing interface
MPPA multi-purpose processor architecture
MPSoC multiprocessor system on a chip
MPU message passing unit

NA network adapter
NI network interface
NoC network on chip
NUMA non-uniform memory access

OPRA output port reservation arbitration
ORT output reservation table
OS operating system

PAR place and route
PCB printed circuit board
PCI peripheral component interconnect
PE processing element
PGAS partitioned global address space
PIO programmed input/output
PMU power management unit
PS packet switching

QoS quality of service

RAM random-access memory
RaR request-and-response
RB ring bus
RE resource element
RGMII reduced gigabit media independent interface
RISC reduced instruction set computer

RMP	resource management policy
RMU	resource management unit
RR	round-robin
RSR	reservation success rate
RT	round-trip
RTP	round-trip packet
SAF	store and forward
SCC	Single-chip Cloud Computer
SDM	spatial division multiplexing
SER	soft error rate
SET	single event transient
SEU	single event upsets
SL	service level
SLN	second layer network
SMU	shared memory unit
SoC	system on a chip
SPMD	single program, multiple data
SRAM	static random-access memory
SSRAM	synchronous static random access memory
STN	static network
TC	transmission control
TCPA	Tightly-Coupled Processor Array
TDM	time division multiplexing
TDMA	time division multiple access
TDN	tile dynamic network
TGFF	task graphs for free
TLM	tile local memory
TMR	triple modular redundancy
TS	time slot
TSV	through-silicon via
UART	universal asynchronous receiver transmitter
UDN	user dynamic network

UMR	Universal Multi-Resource
UPF	unified power format
USB	universal serial bus
VC	virtual channel
VCD	value change dump
VCI	virtual component interface
VCT	virtual cut through
VCU	virtual channel utilization
VFI	voltage-frequency islands
VI	voltage island
VLIW	very long instruction word
VLSI	very-large-scale integration
VN	virtual network
VNCU	virtual network control unit
VNMU	virtual network management unit
WC	worst case
WRR	weighted round-robin
XAUI	10 gigabit media independent interface

Bibliography

[1] AGARWAL, A., C. ISKANDER and R. SHANKAR: *Survey of network on chip (noc) architectures & contributions*. Journal of engineering, Computing and Architecture, 3(1):21–27, 2009.

[2] ANDRZEJEWSKI, M.: *AMBA bus emulation in the Nostrum NoC using best effort communication*. PhD thesis, Citeseer, 2005.

[3] ATTIA, B., W. CHOUCHENE, A. ZITOUNI, A. NOURDIN and R. TOURKI: *Design and implementation of low latency network interface for network on chip*. In *Design and Test Workshop (IDT), 2010 5th International*, pp. 37–42. IEEE, 2010.

[4] AUGONNET, C., S. THIBAULT, R. NAMYST and P.-A. WACRENIER: *StarPU: A unified platform for task scheduling on heterogeneous multicore architectures*. In *European Conference on Parallel Processing*, pp. 863–874. Springer, 2009.

[5] BADAWY, W. and G. JULIEN: *System-on-Chip for Real-Time Applications*. The Springer International Series in Engineering and Computer Science. Springer US, 2003.

[6] BECKER, J., S. FRIEDERICH, J. HEISSWOLF, R. KOENIG and D. MAY: *Hardware prototyping of novel invasive multicore architectures*. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pp. 201–206. IEEE, 2012.

[7] BENINI, L. and G. D. MICHELI: *Networks on chips: a new SoC paradigm*. Computer, 2002.

[8] BERTOZZI, D. and L. BENINI: *Xpipes: a network-on-chip architecture for gigascale systems-on-chip*. Circuits and Systems Magazine, IEEE, 4(2):18–31, 2004.

[9] BERTOZZI, D., A. JALABERT, S. MURALI, S. MEMBER, R. TAMHANKAR, S. MEMBER, S. STERGIOU, S. MEMBER, L. BENINI and G. D. MICHELI: *NoC synthesis flow for customized domain specific multiprocessor Systems-on-Chip*. IEEE Transactions on Parallel and Distributed Systems, 2005.

[10] BHOJWANI, P. and R. MAHAPATRA: *Interfacing cores with on-chip packet-switched networks*. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pp. 382–387. IEEE, 2003.

[11] BHOJWANI, P. and R. N. MAHAPATRA: *Core network interface architecture and latency constrained on-chip communication*. In *Quality Electronic Design, 2006. ISQED'06. 7th International Symposium on*, pp. 6–pp. IEEE, 2006.

[12] BINKERT, N., B. BECKMANN, G. BLACK, S. K. REINHARDT, A. SAIDI, A. BASU, J. HESTNESS, D. R. HOWER, T. KRISHNA, S. SARDASHTI et al.: *The gem5 simulator*. ACM SIGARCH Computer Architecture News, 39(2):1–7, 2011.

[13] BJERREGAARD, T. and S. MAHADEVAN: *A survey of research and practices of network-*

on-chip. ACM Computing Surveys (CSUR), 38(1):1, 2006.

[14] Bjerregaard, T., S. Mahadevan, R. G. Olsen and J. Sparsø: *An OCP compliant network adapter for GALS-based SoC design using the MANGO network-on-chip*. In *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, pp. 171–174. IEEE, 2005.

[15] Bjerregaard, T. and J. Sparso: *A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip*. In *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 1226–1231. IEEE, 2005.

[16] Bolotin, E., I. Cidon, R. Ginosar and A. Kolodny: *QNoC: QoS architecture and design process for network on chip*. Journal of systems architecture, 50(2):105–128, 2004.

[17] Bononi, L. and N. Concer: *Simulation and analysis of network on chip architectures: ring, spidergon and 2D mesh*. In *Proceedings of the conference on Design, automation and test in Europe: Designers' forum*, pp. 154–159. European Design and Automation Association, 2006.

[18] Borkar, S. and A. A. Chien: *The future of microprocessors*. Communications of the ACM, 54(5):67–77, 2011.

[19] Braun, M., S. Buchwald, M. Mohr and A. Zwinkau: *An x10 compiler for invasive architectures*. KIT, Fakultät für Informatik, 2012.

[20] Carara, E., G. M. Almeida, G. Sassatelli and F. G. Moraes: *Achieving composability in NoC-based MPSoCs through QoS management at software level*. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6. IEEE, 2011.

[21] Chang, J., Y. Jongsu and K. JunSeong: *Design a switch wrapper for SNA on-chip-network*. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 89(6):1615–1621, 2006.

[22] Charles, P., C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun and V. Sarkar: *X10: an object-oriented approach to non-uniform cluster computing*. Acm Sigplan Notices, 40(10):519–538, 2005.

[23] Chen, X., Z. Lu, A. Jantsch and S. Chen: *Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller*. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 39–44, 2010.

[24] Cheng, L., J. B. Carter and D. Dai: *An adaptive cache coherence protocol optimized for producer-consumer sharing*. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 328–339. IEEE, 2007.

[25] Coporation, A.: *Avalon interface specifications*, 2005.

[26] Dall'Osso, M., G. Biccari, L. Giovannini, D. Bertozzi and L. Benini: *Xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs*. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pp. 45–48. IEEE, 2012.

[27] Dally, W. J.: *Virtual-channel flow control*. Parallel and Distributed Systems, IEEE

Transactions on, 3(2):194–205, 1992.

[28] DAMODARAN, P. P., S. WALLENTOWITZ and A. HERKERSDORF: *Distributed cooperative shared last-level caching in tiled multiprocessor system on chip*. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–4. IEEE, 2014.

[29] DAMODARAN, P. P. M., A. ZAIB, T. WILD, S. WALLENTOWITZ and A. HERKERSDORF: *Sharer Status-based Caching in tiled Multiprocessor Systems-on-Chip*. In *High Performance Computing (HPC), 2015*, 2015.

[30] DAS, A., M. SCHUCHHARDT, N. HARDAVELLAS, G. MEMIK and A. CHOUDHARY: *Dynamic directories: A mechanism for reducing on-chip interconnect power in multicores*. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 479–484. EDA Consortium, 2012.

[31] DAS, R., O. MUTLU, T. MOSCIBRODA and C. R. DAS: *Application-aware prioritization mechanisms for on-chip networks*. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 280–291. IEEE, 2009.

[32] DAYA, B. K., C.-H. O. CHEN, S. SUBRAMANIAN, W.-C. KWON, S. PARK, T. KRISHNA, J. HOLT, A. P. CHANDRAKASAN and L.-S. PEH: *SCORPIO: a 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering*. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 25–36. IEEE, 2014.

[33] DE MICHELI, G. and L. BENINI: *Networks on chips: technology and tools*. Academic Press, 2006.

[34] EBRAHIMI, M., M. DANESHTALAB, N. SREEJESH, P. LILJEBERG and H. TENHUNEN: *Efficient network interface architecture for network-on-chips*. Proc. of 27th IEEE Norchip, pp. 1–4, 2009.

[35] ESMAEILZADEH, H., E. BLEM, R. S. AMANT, K. SANKARALINGAM and D. BURGER: *Dark silicon and the end of multicore scaling*. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pp. 365–376. IEEE, 2011.

[36] FARUQUE, A., M. ABDULLAH, T. EBI and J. HENKEL: *Run-time adaptive on-chip communication scheme*. In *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pp. 26–31. IEEE, 2007.

[37] FERRANTE, A., S. MEDARDONI and D. BERTOZZI: *Network interface sharing techniques for area optimized NoC architectures*. In *Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on*, pp. 10–17. IEEE, 2008.

[38] FLICH, J. and D. BERTOZZI: *Designing network on-chip architectures in the nanoscale era*. CRC Press, 2010.

[39] FLYNN, D.: *AMBA: enabling reusable on-chip designs*. Micro, IEEE, 17(4):20–27, 1997.

[40] FREEMAN, C. and L. SOETE: *The economics of industrial innovation*. Psychology Press, 1997.

[41] FRIEDERICH, S., J. HEISSWOLF and J. BECKER: *Hardware/software debugging of large scale many-core architectures*. In *Integrated Circuits and Systems Design (SBCCI), 2014*

27th Symposium on, pp. 1–7. IEEE, 2014.

[42] FRIEDERICH, S., J. HEISSWOLF, D. MAY and J. BECKER: *Hardware prototyping and software debugging of multi-core architectures*.

[43] GAISLER, A.: *Leon3 processor*. Nanoscale Integration and Modeling (NIMO) Group, 2010.

[44] GAISLER, J., E. CATOVIC, M. ISOMAKI, K. GLEMBO and S. HABINC: *GRLIB IP core user manual*. Gaisler research, 2007.

[45] GEBALI, F., H. ELMILIGI and M. W. EL-KHARASHI: *Networks-on-chips: theory and practice*. CRC press, 2011.

[46] GEER, D.: *Chip makers turn to multicore processors*. Computer, 38(5):11–13, 2005.

[47] GERNDT, M., F. HANNIG, A. HERKERSDORF, A. HOLLMANN, M. MEYER, S. ROLOFF, J. WEIDENDORFER, T. WILD and A. ZAIB: *An integrated simulation framework for invasive computing*. In *Specification and Design Languages (FDL), 2012 Forum on*, pp. 209–216. IEEE, 2012.

[48] GOOSSENS, K., J. DIELISSEN and A. RADULESCU: *Æthereal network on chip: concepts, architectures, and implementations*. Design & Test of Computers, IEEE, 22(5):414–421, 2005.

[49] GOOSSENS, K., P. WIELAGE, A. PEETERS and J. VAN MEERBERGEN: *Networks on silicon: Combining best-effort and guaranteed services*. In *date*, p. 0423. IEEE, 2002.

[50] GRAPHICS, M.: *ModelSim*, 2007.

[51] GROT, B., J. HESTNESS, S. W. KECKLER and O. MUTLU: *Kilo-NOC: a heterogeneous network-on-chip architecture for scalability and service guarantees*. ACM SIGARCH Computer Architecture News, 39(3):401–412, 2011.

[52] HANNIG, F., S. ROLOFF, G. SNELTING, J. TEICH and A. ZWINKAU: *Resource-aware programming and simulation of MPSoC architectures through extension of X10*. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, pp. 48–55. ACM, 2011.

[53] HEISSWOLF, J.: *A Scalable and Adaptive Network on Chip for Many-Core Architectures*. PhD thesis, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2014, 2014.

[54] HEISSWOLF, J., S. FRIEDERICH, L. MASING, A. WEICHSLGARTNER, A. ZAIB, C. STEIN, M. DUDEN, J. TEICH, A. HERKERSDORF and J. BECKER: *A Novel NoC-Architecture for Fault Tolerance and Power Saving*. In *Proceedings of the second International Workshop on Multi-Objective Many-Core Design (MOMAC) in conjunction with International Conference on Architecture of Computing Systems (ARCS)*, 2016.

[55] HEISSWOLF, J., R. KÖNIG and J. BECKER: *A scalable NoC router design providing QoS support using weighted round robin scheduling*. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pp. 625–632. IEEE, 2012.

[56] HEISSWOLF, J., A. WEICHSLGARTNER, A. ZAIB, S. FRIEDERICH, L. MASING, C. STEIN, M. DUDEN, R. KLOPFER, J. TEICH, T. WILD et al.: *Fault-tolerant communication in invasive networks on chip*. In *Adaptive Hardware and Systems (AHS), 2015*

NASA/ESA Conference on, pp. 1–8. IEEE, 2015.

[57] HEISSWOLF, J., A. ZAIB, A. WEICHSLGARTNER, M. KARLE, M. SINGH, T. WILD, J. TEICH, A. HERKERSDORF and J. BECKER: *The invasive network on chip-a multi-objective many-core communication infrastructure*. In *Architecture of Computing Systems (ARCS), 2014 27th International Conference on*, pp. 1–8. VDE, 2014.

[58] HEISSWOLF, J., A. ZAIB, A. WEICHSLGARTNER, R. KÖNIG, T. WILD, J. TEICH, A. HERKERSDORF and J. BECKER: *Hardware-assisted decentralized resource management for networks on chip with qos*. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pp. 234–241. IEEE, 2012.

[59] HEISSWOLF, J., A. ZAIB, A. WEICHSLGARTNER, R. KÖNIG, T. WILD, J. TEICH, A. HERKERSDORF and J. BECKER: *Virtual networks–distributed communication resource management*. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 6(2):8, 2013.

[60] HEISSWOLF, J., A. ZAIB, A. ZWINKAU, S. KOBBE, A. WEICHSLGARTNER, J. TEICH, J. HENKEL, G. SNELTING, A. HERKERSDORF and J. BECKER: *CAP: Communication aware programming*. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pp. 1–6. IEEE, 2014.

[61] HENKEL, J.: *Closing the SoC design gap*. Computer, 36(9):119–121, 2003.

[62] HENKEL, J., L. BAUER, M. HÜBNER and A. GRUDNITSKY: *i-Core: A run-time adaptive processor for embedded multi-core systems*. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2011)*, 2011.

[63] HENKEL, J., A. HERKERSDORF, L. BAUER, T. WILD, M. HÜBNER, R. K. PUJARI, A. GRUDNITSKY, J. HEISSWOLF, A. ZAIB, B. VOGEL et al.: *Invasive manycore architectures.*. In *ASP-DAC*, pp. 193–200, 2012.

[64] HERKERSDORF, A., J. PAUL, R. K. PUJARI, W. STECHELE, S. WALLENTOWITZ, T. WILD and A. ZAIB: *Potentials and Challenges for Multi-Core Processors in Robotic Applications.*. In *GI-Jahrestagung*, pp. 2749–2764, 2013.

[65] HILL, M. D. and M. R. MARTY: *Amdahl's law in the multicore era*. Computer, (7):33–38, 2008.

[66] HILTON, C. and B. NELSON: *PNoC: a flexible circuit-switched NoC for FPGA-based systems*. IEE Proceedings-Computers and Digital Techniques, 153(3):181–188, 2006.

[67] HOWARD, J., S. DIGHE and Y. H. ET. AL.: *A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS*. In *ISSCC*, 2010.

[68] HU, J. and R. MARCULESCU: *Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures*. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 688–693. IEEE, 2003.

[69] HU, J. and R. MARCULESCU: *DyAD: smart routing for networks-on-chip*. In *Proceedings of the 41st annual Design Automation Conference*, pp. 260–263. ACM, 2004.

[70] IBM: *CoreConnect Bus Architecture*, June 2015.

[71] INFINEON: *Automotive - Innovative automotive electronics by Infineon*, Jan. 2016.

[72] ISSENIN, I., E. BROCKMEYER, M. MIRANDA and N. DUTT: *Data reuse analysis technique for software-controlled memory hierarchies*. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 1, pp. 202–207. IEEE, 2004.

[73] JANTSCH, A., H. TENHUNEN et al.: *Networks on chip*, vol. 396. Springer, 2003.

[74] JEFFERS, J. and J. REINDERS: *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[75] JOVEN MURILLO, J., J. CARRABINA I BORDOLL et al.: *HW-sw components for parallel embedded computing on noc-based mpsocs*. Universitat Autònoma de Barcelona,, 2010.

[76] JUNG, E. B., H. W. CHO, N. PARK and Y. H. SONG: *Sona: An on-chip network for scalable interconnection of amba-based ips*. In *Computational Science–ICCS 2006*, pp. 244–251. Springer, 2006.

[77] KATEVENIS, M., S. SIDIROPOULOS and C. COURCOUBETIS: *Weighted round-robin cell multiplexing in a general-purpose ATM switch chip*. Selected Areas in Communications, IEEE Journal on, 9(8):1265–1279, 1991.

[78] KAVADIAS, S. G., M. G. KATEVENIS, M. ZAMPETAKIS and D. S. NIKOLOPOULOS: *On-chip Communication and Synchronization Mechanisms with Cache-integrated Network Interfaces*. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, 2010.

[79] KAVALDJIEV, N., G. J. SMIT, P. T. WOLKOTTE and P. G. JANSEN: *Providing QoS guarantees in a NoC by virtual channel reservation*. In *Reconfigurable Computing: Architectures and Applications*, pp. 299–310. Springer, 2006.

[80] KAVALDJIEV, N. K., G. J. M. SMIT and P. G. JANSEN: *A virtual channel router for on-chip networks*. 2004.

[81] KISSLER, D., F. HANNIG, A. KUPRIYANOV and J. TEICH: *A highly parameterizable parallel processor array architecture*. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pp. 105–112. IEEE, 2006.

[82] KOBBE, S., L. BAUER, D. LOHMANN, W. SCHRÖDER-PREIKSCHAT and J. HENKEL: *DistRM: distributed resource management for on-chip many-core systems*. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 119–128. ACM, 2011.

[83] KUMAR, A., L.-S. PEH, P. KUNDU and N. K. JHA: *Express virtual channels: towards the ideal interconnection fabric*. In *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 150–161. ACM, 2007.

[84] KUMAR, R., V. ZYUBAN and D. M. TULLSEN: *Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling*. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pp. 408–419. IEEE, 2005.

[85] LAMPRET, D., C.-M. CHEN, M. MLINAR, J. RYDBERG, M. ZIV-AV, C. ZIOMKOWSKI, G. MCGARY, B. GARDNER, R. MATHUR and M. BOLADO: *Openrisc 1000 architecture manual*. Rev, 1:15, 2007.

[86] LANKES, A., T. WILD, A. HERKERSDORF, S. SONNTAG and H. REINIG: *Comparison of deadlock recovery and avoidance mechanisms to approach message dependent deadlocks in*

on-chip networks. In *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*, pp. 17–24. IEEE, 2010.

[87] LEE, S. E., J. H. BAHN, Y. S. YANG and N. BAGHERZADEH: *A generic network interface architecture for a networked processor array (NePA)*. In *Architecture of Computing Systems–ARCS 2008*, pp. 247–260. Springer, 2008.

[88] LI, B., L. ZHAO, R. IYER, L.-S. PEH, M. LEDDIGE, M. ESPIG, S. E. LEE and D. NEWELL: *CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs*. Journal of Parallel and Distributed Computing, 71(5):700–713, 2011.

[89] LORINCZ, K., B.-R. CHEN, J. WATERMAN, G. WERNER-ALLEN and M. WELSH: *Resource aware programming in the pixie os*. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pp. 211–224. ACM, 2008.

[90] MEREU, G.: *Conception, Analysis, Design and Realization of a Multi-socket Network-on-Chip Architecture and of the Binary Translation support for VLIW core targeted to Systems-on-Chip*, 2007.

[91] MICROELECTRONICS, S.: *STBus interconnect*.

[92] MICROTEC, A.: *A°AC OpenPIC System on Chip library*, Nov. 2015.

[93] MILLBERG, M., E. NILSSON, R. THID and A. JANTSCH: *Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip*. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 2, pp. 890–895. IEEE, 2004.

[94] MILLBERG, M., E. NILSSON, R. THID, S. KUMAR and A. JANTSCH: *The Nostrum backbone-a communication protocol stack for networks on chip*. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pp. 693–696. IEEE, 2004.

[95] MONCHIERO, M., G. PALERMO, C. SILVANO and O. VILLA: *Efficient synchronization for embedded on-chip multiprocessors*. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 14(10):1049–1062, 2006.

[96] MUBEEN, S.: *Evaluation of source routing for mesh topology network on chip platforms*. 2009.

[97] MULLINS, R., A. WEST and S. MOORE: *Low-latency virtual-channel routers for on-chip networks*. In *ACM SIGARCH Computer Architecture News*, vol. 32, p. 188. IEEE Computer Society, 2004.

[98] MURALI, S., L. BENINI and G. DE MICHELI: *Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees*. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, vol. 1, pp. 27–32. IEEE, 2005.

[99] MURALI, S., P. MELONI, F. ANGIOLINI, D. ATIENZA, S. CARTA, L. BENINI, G. DE MICHELI and L. RAFFO: *Designing message-dependent deadlock free networks on chips for application-specific systems on chips*. In *Very Large Scale Integration, 2006 IFIP International Conference on*, pp. 158–163. IEEE, 2006.

[100] MURRAY, J., P. WETTIN, P. P. PANDE and B. SHIRAZI: *Sustainable Wireless Network-on-Chip Architectures*. Morgan Kaufmann, 2016.

[101] NAVABI, Z.: *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[102] NIKHIL, R.: *Bluespec System Verilog: efficient, correct RTL from high level specifications*. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pp. 69–70. IEEE, 2004.

[103] NITZBERG, B. and V. LO: *Distributed shared memory: A survey of issues and algorithms*. Computer, 24(8):52–60, 1991.

[104] OECHSLEIN, B., J. SCHEDEL, J. KLEINÖDER, L. BAUER, J. HENKEL, D. LOHMANN and W. SCHRÖDER-PREIKSCHAT: *OctoPOS: A parallel operating system for invasive computing*. In *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*, pp. 9–14, 2011.

[105] OGRAS, U. Y. and R. MARCULESCU: *Application-specific network-on-chip architecture customization via long-range link insertion*. In *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, pp. 246–253. IEEE, 2005.

[106] OGRAS, U. Y. and R. MARCULESCU: *" It's a small world after all": NoC performance optimization via long-range link insertion*. IEEE Transactions on very large scale integration (VLSI) systems, 14(7):693–706, 2006.

[107] OGRAS, U. Y. and R. MARCULESCU: *Prediction-based flow control for network-on-chip traffic*. In *Proceedings of the 43rd annual design automation conference*, pp. 839–844. ACM, 2006.

[108] OLSEN, R. G.: *OCP based adapter for network-on-chip*. PhD thesis, Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark, 2005.

[109] OPENCORES: *SoC Interconnection: Wishbone*, June 2015.

[110] OUSTERHOUT, J. K.: *Tcl: An embeddable command language*. Citeseer, 1989.

[111] PASRICHA, S. and N. DUTT: *On-chip communication architectures: system on chip interconnect*. Morgan Kaufmann, 2010.

[112] PASTRNAK, M., P. H. DE WITH and J. VAN MEERBERGEN: *Realization of qos management using negotiation algorithms for multiprocessor noc*. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pp. 4–pp. IEEE, 2006.

[113] PETROT, F., A. GREINER and P. GOMEZ: *On cache coherency and memory consistency issues in NoC based shared memory multiprocessor SoC architectures*. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pp. 53–60. IEEE, 2006.

[114] POLETTI, F., A. POGGIALI, D. BERTOZZI, L. BENINI, P. MARCHAL, M. LOGHI and M. PONCINO: *Energy-efficient multiprocessor systems-on-chip for embedded computing: Exploring programming models and their architectural support*. IEEE Trans. Computers, 56(5):606–621, 2007.

[115] PORRMANN, M., M. PURNAPRAJNA and C. PUTTMANN: *Self-optimization of MPSoCs targeting resource efficiency and fault tolerance*. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pp. 467–473. IEEE, 2009.

[116] PUJARI, R. K., T. WILD and A. HERKERSDORF: *A hardware-based multi-objective thread mapper for tiled manycore architectures*. In *Computer Design (ICCD), 2015 33rd*

IEEE International Conference on, pp. 459–462. IEEE, 2015.

[117] PUJARI, R. K., T. WILD, A. HERKERSDORF, B. VOGEL and J. HENKEL: *Hardware assisted thread assignment for RISC based MPSoCs in invasive computing*. In *Integrated Circuits (ISIC), 2011 13th International Symposium on*, pp. 106–109. IEEE, 2011.

[118] RADULESCU, A., J. DIELISSEN, S. G. PESTANA, O. P. GANGWAL, E. RIJPKEMA, P. WIELAGE and K. GOOSSENS: *An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 24(1):4–17, 2005.

[119] RAHMANI, A.-M., A. AFZALI-KUSHA and M. PEDRAM: *A novel synthetic traffic pattern for power/performance analysis of network-on-chips using negative exponential distribution*. Journal of Low Power Electronics, 5(3):396–405, 2009.

[120] RAMEY, C.: *Tile-gx100 manycore processor: Acceleration interfaces and architecture*. In *Proceedings of the 23th Hot Chips Symposium*, 2011.

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

[122] SANGIOVANNI-VINCENTELLI, A.: *Defining platform-based design*. EEDesign of EE-Times, 2002.

[123] SCHALLER, R. R.: *Moore's law: past, present and future*. Spectrum, IEEE, 34(6):52–59, 1997.

[124] SEMERIA, C.: *Supporting differentiated service classes: queue scheduling disciplines*. Juniper networks, pp. 11–14, 2001.

[125] SHIPILOV, D.: *Design and implementation of the resource-network interface for networks-on-chip*. PhD thesis, Citeseer, 2004.

[126] SPECIFICATION, A.: *ARM, Ltd*, 2001.

[127] SRINIVASAN, J.: *An overview of static power dissipation*. CiteSeer public search engine and digital libraries for scientific and academic papers in the fields of computer and information science, pp. 1–7, 2011.

[128] STINGASFEYD: *Microbots: Automation Revolution Continues with Miniaturized Electronics*, Feb. 2014.

[129] STINGASFEYD: *Did Moore's law calculate correctly?*, Apr. 2015.

[130] SYNOPSYS: *End-to-End Prototyping*, Oct. 2015.

[131] TEICH, J., J. HENKEL, A. HERKERSDORF, D. SCHMITT-LANDSIEDEL, W. SCHRÖDER-PREIKSCHAT and G. SNELTING: *Invasive computing: An overview*. In *Multiprocessor System-on-Chip*, pp. 241–268. Springer, 2011.

[132] TOTA, S. V., M. R. CASU, M. R. ROCH, L. ROSTAGNO and M. ZAMBONI: *MEDEA: a hybrid shared-memory/message-passing multiprocessor NoC-based architecture*. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pp. 45–50. IEEE, 2010.

[133] VAN DER TOL, E. B. and E. G. JASPERS: *Mapping of MPEG-4 decoding on a flexible architecture platform*. In *Electronic Imaging 2002*, pp. 1–13. International Society for Optics and Photonics, 2001.

[134] VARGHESE, A., B. EDWARDS, G. MITRA and A. P. RENDELL: *Programming the Adapteva Epiphany 64-core Network-on-chip Coprocessor*. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pp. 984–992. IEEE, 2014.

[135] WALLENTOWITZ, S., A. LANKES, A. ZAIB, T. WILD and A. HERKERSDORF: *A framework for open tiled manycore system-on-chip*. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 535–538. IEEE, 2012.

[136] WALLENTOWITZ, S., T. WILD and A. HERKERSDORF: *HW-OSQM: reducing the impact of event signaling by hardware-based operating system queue manipulation*. In *International Conference on Architecture of Computing Systems*, pp. 280–291. Springer, 2013.

[137] WEICHSLGARTNER, A., J. HEISSWOLF, A. ZAIB, T. WILD, A. HERKERSDORF, J. BECKER and J. TEICH: *Position Paper: Towards Hardware-Assisted Decentralized Mapping of Applications for Heterogeneous NoC Architectures*. In *Architecture of Computing Systems. Proceedings, ARCS 2015-The 28th International Conference on*, pp. 1–4. VDE, 2015.

[138] WIKIPEDIA: *List of Intel Core i7 microprocessors — Wikipedia, The Free Encyclopedia*, 2015. [Online; accessed 21-May-2015].

[139] WIKIPEDIA: *Distributed shared memory — Wikipedia, The Free Encyclopedia*, 2016. [Online; accessed 4-August-2016].

[140] WIKIPEDIA: *Intel Atom — Wikipedia, The Free Encyclopedia*, 2016. [Online; accessed 17-November-2016].

[141] WIKIPEDIA: *Partitioned global address space — Wikipedia, The Free Encyclopedia*, 2016. [Online; accessed 4-August-2016].

[142] YELICK, K., D. BONACHEA, W.-Y. CHEN, P. COLELLA, K. DATTA, J. DUELL, S. L. GRAHAM, P. HARGROVE, P. HILFINGER, P. HUSBANDS et al.: *Productivity and performance using partitioned global address space languages*. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pp. 24–32. ACM, 2007.

[143] ZAIB, A.: *Intelligent Hardware Support for Hybrid Message Passing in Tiled Multicore Architectures*, 2010.

[144] ZAIB, A., J. HEISSWOLF, A. WEICHSLGARTNER, T. WILD, J. TEICH, J. BECKER and A. HERKERSDORF: *AUTO-GS: Self-Optimization of NoC Traffic through Hardware Managed Virtual Connections*. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pp. 761–768. IEEE, 2013.

[145] ZAIB, A., J. HEISSWOLF, A. WEICHSLGARTNER, T. WILD, J. TEICH, J. BECKER and A. HERKERSDORF: *Network Interface with Task Spawning Support for NoC-Based DSM Architectures*. In *Architecture of Computing Systems–ARCS 2015*, pp. 186–198. Springer, 2015.

[146] ZAIB, A., P. RAJU, T. WILD and A. HERKERSDORF: *A Layered Modeling and Sim-*

ulation Approach to investigate Resource-aware Computing in MPSoCs. arXiv preprint arXiv:1405.2917, 2014.

[147] ZAIB, A., T. WILD, A. HERKERSDORF, J. HEISSWOLF, J. BECKER, A. WEICHSLGARTNER and J. TEICH: *Efficient Task Spawning for Shared Memory and Message Passing in Many-core Architectures*. Journal of Systems Architecture, 2017.

[148] ZHU, W., V. C. SREEDHAR, Z. HU and G. R. GAO: *Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures*. In *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 35–45. ACM, 2007.

[149] ZIMMER, C. and F. MUELLER: *Low contention mapping of real-time tasks onto tilepro 64 core processors*. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pp. 131–140. IEEE, 2012.

[150] ZIMMERMANN, H.: *OSI reference model–The ISO model of architecture for open systems interconnection*. Communications, IEEE Transactions on, 28(4):425–432, 1980.