Phạm Quốc Cường

HYBRID INTERCONNECT DESIGN FOR HETEROGENEOUS HARDWARE ACCELERATORS

HYBRID INTERCONNECT DESIGN FOR HETEROGENEOUS HARDWARE ACCELERATORS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K. C. A. M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op
dinsdag 14 April 2015 om 12:30 uur

door

Cuong PHAM-QUOC

Master of Engineering in Computer Science
Ho Chi Minh City University of Technology - HCMUT, Vietnam
geboren te Tien Giang, Vietnam.

This dissertation has been approved by the

Promotor: Prof.dr. K.L.M Bertels
Copromotor: Dr.ir. Z. Al-Ars

Composition of the doctoral committee:

Rector Magnificus	voorzitter
Prof.dr. K.L.M Bertels	Technische Universiteit Delft, promotor
Dr.ir. Z. Al-Ars	Technische Universiteit Delft, copromotor
Independent members:	
Prof.dr. E. Charbon	Technische Universiteit Delft
Prof.dr.-ing. J. Becker	Karlsruhe Institute of Technology
Prof.dr. A.V. Dinh-Duc	Vietnam National University - Ho Chi Minh City
Prof.dr. Luigi Carro	Universidade Federal do Rio Grande do Sul
Dr. F. Silla	Universitat Politècnica de València
Prof.dr.ir. A.-J van der Veen	Technische Universiteit Delft, reservelid

Keywords: Hybrid interconnect, hardware accelerators, data communication, quantitative data usage, automated design.

ISBN 978-94-6186-448-2

Cover design: Cuong Pham-Quoc
Printed in The Netherlands

To my wife and my son

ABSTRACT

Heterogeneous multicore systems are becoming increasingly important as the need for computation power grows, especially when we are entering into the big data era. As one of the main trends in heterogeneous multicore, hardware accelerator systems provide application specific hardware circuits and are thus more energy efficient and have higher performance than general purpose processors, while still providing a large degree of flexibility. However, system performance dose not scale when increasing the number of processing cores due to the communication overhead which increases greatly with the increasing number of cores. Although data communication is a primary anticipated bottleneck for system performance, the interconnect design for data communication among the accelerator kernels has not been well addressed in hardware accelerator systems. A simple bus or shared memory is usually used for data communication between the accelerator kernels. In this dissertation, we address the issue of interconnect design for heterogeneous hardware accelerator systems.

Evidently, there are dependencies among computations, since data produced by one kernel may be needed by another kernel. Data communication patterns can be specific for each application and could lead to different types of interconnect. In this dissertation, we use detailed data communication profiling to design an optimized hybrid interconnect that provides the most appropriate support for the communication pattern inside an application while keeping the hardware resource usage for the interconnect minimal. Firstly, we propose a heuristic-based approach that takes application data communication profiling into account to design a hardware accelerator system with a custom interconnect. A number of solutions are considered including crossbar-based shared local memory, direct memory access (DMA) supporting parallel processing, local buffers, and hardware duplication. This approach is mainly useful for embedded system where the hardware resources are limited. Secondly, we propose an automated hybrid interconnect design using data communication profiling to define an optimized interconnect for accelerator kernels of a generic hardware accelerator system. The hybrid interconnect consists of a network-on-chip (NoC),

shared local memory, or both. To minimize hardware resource usage for the hybrid interconnect, we also propose an adaptive mapping algorithm to connect the computing kernels and their local memories to the proposed hybrid interconnect. Thirdly, we propose a hardware accelerator architecture to support streaming image processing. In all presented approaches, we implement the approach using a number of benchmarks on relevant reconfigurable platforms to show their effectiveness. The experimental results show that our approaches not only improve system performance but also reduce overall energy consumption compared to the baseline systems.

ACKNOWLEDGMENTS

It is not easy to write this last part of the dissertation, but this is an exciting period because it lets me take a careful look at the whole last four years, starting from 2011. First, I would like to thank the Vietnam International Education Development (VIED) for their funding. Without this funding, I would not have been in the Netherlands.

I would like to express special appreciation and thanks to my promoter, Prof. Dr. Koen Bertels, who had a difficult decision, but a successful one, when accepting me as his Ph.D. student in 2011. At that time, my spoken English was not very good but he tried very hard to understand our Skype-based discussion. During my time at the Computer Engineering Lab, he has introduced me to so many great ideas and has given me freedom to do my research. Koen, without you, I would have had no chance to write this dissertation. Another significant appreciation and thanks are given to my daily supervisor, but he always says that I am his friend, Dr.Ir. Zaid Al-Ars, who has guided me a lot not only in doing research but also in writing a paper. Zaid, I can never forget the many hours you have spent correcting my papers. Without you, I would have no publication and, of course, no dissertation. Besides these two great persons, I would like to say thank you to Veronique from Valorisation Center - TUDelft, Lidwina - CE secretary, and Eef and Erik - CE system administrators, for their support. I would like to thank my colleagues, Razvan, for your DWARV compiler and, Vlad, for the Molen platform upon which I have conducted the experiments. Thank you, Ernst, for your time translating my abstract and my proposition into Dutch.

I need to say thank you to Prof. Dr. Anh-Vu Dinh-Duc. This is the third time I have written his name in my thesis. The first and the second times were as my supervisor while this time is as a committee member. He has been there at many steps of my learning journey. I also appreciate all the committee members' time and the remarks they gave me.

Life is not only doing research. Without relaxing time and parties, we have no energy and no ideas. So, thank you to the ANCB group, a group of Vietnamese students, for the very enjoyable parties. Those parties and relaxing time helped

me refresh my mind after the tiring working days. I am sure that I cannot say thank you to everybody who has supported me during the last four years because it would take a hundred pages, but I am also sure that I will never forget. Let me keep your kindness in my mind.

I am extremely grateful for my family and my wife's family, especially my father in law and my mother in law who have helped me to take care of my son when I could not be at home. Without you, I would not have had the peace of mind to do my work.

Last but most importantly, I would like to say thank you so much my wife and my son. You raise me up, and you make me stronger. Without your love and your support, I cannot do anything. Our family is going to reunite in the next couple of months after a long period of connecting together through a "hybrid interconnect" - a combination of video-calls, telephone calls, emails, social networks, and traveling.

Phạm Quốc Cường
Delft, April 2015

CONTENTS

LIST OF FIGURES

List of Tables

1

INTRODUCTION

WITH the rapid development of technology, more and more transistors are integrated on a single chip. Today, it is possible to integrate more than 20 billion transistors [Leibson, 2014] into one system (announced by Xilinx in May 2014). However, the more transistors are integrated into a system; the more challenges need to be addressed such as power consumption, thermal emission and memory access bottleneck. Homogeneous and heterogeneous multicore systems were introduced to utilize such large numbers of transistor efficiently.

A generic multicore architecture can be seen as a multiprocessor system in which multiple processing elements (PEs) (also called computational cores) and a memory system are tightly connected together through a communication infrastructure (interconnect). Besides these three main components (PEs, memory system and communication infrastructure), a multicore architecture typically contains other components such as I/O, timer, etc.

- *Processing elements*: In a multicore system, PEs have various types ranging from general purpose processors to *Intellectual Property* (IP) cores. PEs may support either software tasks or hardware tasks. Software tasks can be performed in instruction set processors such as PowerPC, ARM, etc; while hardware tasks can be executed in hardware cores such as reconfigurable logic or dedicated IP cores. Based on the type of PEs, multicore architectures are classified into two classes called homogeneous and heterogeneous architecture. In the homogeneous multicore architecture (Fig-

ure 1.1(a)), all PEs are identical. PEs in the heterogeneous multicore architecture (Figure 1.1(b)) are different types such as general purpose processors, hardware accelerators, dedicated IP cores, etc. Each PE can efficiently and effectively process specific application tasks.

- *Memory system*: Like other systems, memory in a multicore system contains application data as well as instruction data for instruction set processors. Based on the hierarchy of the memory modules, there are two types of memory systems: shared memory and distributed memory. In shared memory multicore systems, all PEs share the same memory resource (Figure 1.2(a)); therefore, any change made by one PE is visible for all other PEs in the system. In distributed memory multicore systems, each PE has its own memory resource (Figure 1.2(b)); therefore, one PE cannot directly read or write the memory of another PE. Some systems have a hybrid memory architecture of both shared and distributed memory. This type of memory architecture is referred to as heterogeneous memory.

- *Communication infrastructure*: The communication infrastructure component in a multicore system (also called interconnect) is a predefined backbone upon which other components are connected together. The communication infrastructure provides a medium for data exchange among PEs as well as between PEs and memory modules in multicore architectures. In modern digital system design, the communication infrastructure is a primary limitation in performance of the whole system [Dally and Towles, 2007]. Therefore, interconnect is a key factor in the digital system design.

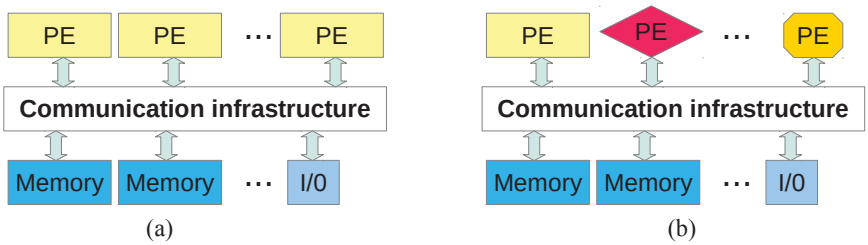


Figure 1.1: (a) Homogeneous multicore; (b) Heterogeneous multicore

Compared to homogeneous multicore systems, heterogeneous multicore systems offer more computation power and efficient energy consumption [Kumar

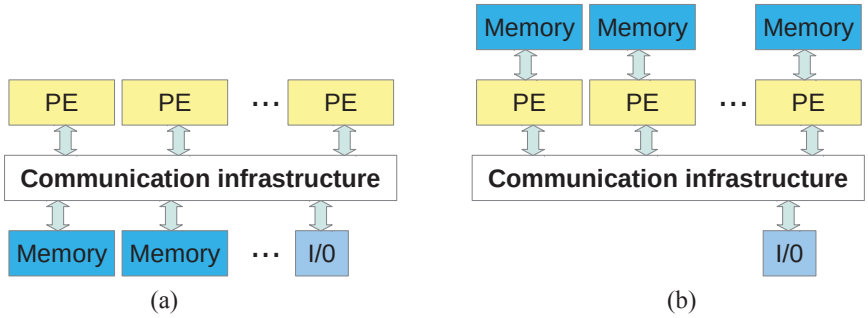


Figure 1.2: (a) Shared memory; (b) Distributed memory

et al., 2005] because of the efficiency of specialized cores for specific tasks. In the past years, a trend towards heterogeneous on chip platforms can be observed. Intel's Atom E6x5C Processor [Intel, 2010] uses multiple RISC cores in combination with an FPGA fabric provided by Altera. Another widely known heterogeneous system is IBM Cell Broadband Engine which contains one PowerPC processor and eight Synergistic Processor Elements [IBM, 2009]. Modern mobile devices are also based on heterogeneous system-on-chips (SoCs) combining CPUs, GPUs and specialized accelerators on a single chip.

As one of the main trends in heterogeneous multicore, hardware accelerator systems provide application specific hardware circuits and are thus more energy efficient and have higher performance than general purpose processors while still providing a significant degree of flexibility. Hardware accelerator systems have been considered as a main approach to continue performance improvement in the future [Borkar and Chien, 2011; Esmaeilzadeh et al., 2011]. They are increasingly popular both in the embedded system domain as well as in high performance computing. This technology has been popular for quite a while in academia [Vassiliadis et al., 2004; Voros et al., 2013] and more and more in the industry championed by companies such as Maxeler [Pell and Mencer, 2011], Convey [Convey Computer, 2012], IBM Power 8 [Stuecheli, 2013], Microsoft Catapult [Putnam et al., 2014], etc. In such systems, there is often one general purpose processor that functions as a host processor and one or more hardware accelerators that function as co-processors to speed-up the processing of special kernels of the application running on the host. Examples of application domains using such accelerators are image processing [Acasandrei and Barriga, 2013; Cong and Zou, 2009; Hung et al., 1999], video-based driver assistance [Claus and Stechele, 2010; Liu et al., 2011], bio-informatics applications [Heideman et al.,

2012; Ishikawa et al., 2012; Sarkar et al., 2010], SAT problem solver [Yuan et al., 2012], etc. However, the main problem of those systems is the communication and data movement overhead they impose [Nilakantan et al., 2013].

1.1. PROBLEM OVERVIEW

The need for computation power grows especially when we are entering into the big data era, where the amount of data grows faster than the capabilities of processing technology. One solution is to increase the number of processing cores especially hardware accelerator kernels for computationally intensive functions. However, the system performance does not scale in this approach due to the communication overhead which increases greatly with the increasing number of cores [Diamond et al., 2011]. In this dissertation, we address the issue of interconnect design for the heterogeneous multicore systems while mainly focusing on hardware accelerator systems.

Interconnect in a multicore system plays an important role because data is exchanged between all components, typically between PEs and memory modules, using the interconnect. Interconnect design is one of the two open issues along with programming model in multicore system design [Rutzig, 2013]. Although data communication is a primary anticipated bottleneck for system performance [Dally and Towles, 2007; Kavadias et al., 2010; Orduña et al., 2004], the interconnect design for data communication among the accelerator kernels has not been well addressed in hardware accelerator systems. A simple bus or shared memory is usually used for data communication between the host and the kernels[1] as well as among the kernels. Although buses have some certain advantages such as low cost and simplicity, they become inefficient when the number of cores rises [Guerrier and Greiner, 2000]. Crossbars have been used to connect the PEs in some systems such as in [Cong and Xiao, 2013; Johnson and Nawathe, 2007]. Despite the high performance, crossbars suffer from high area cost and poor scalability [Rutzig, 2013]. Networks on Chips (NoCs) [Benini and De Micheli, 2002] have been proposed as an efficient communication infrastructure in large systems to allow parallel communication and to increase the scalability compared to buses. However, the major drawbacks of NoCs are the increased latency and implementation costs [Guerrier and Greiner, 2000]. Shared memory also has its own disadvantages such as restricted access due to the finite

[1] In this work, we use the terminology *kernel* to refer to a dedicated hardware module/circuit that accelerates the processing of a computationally intensive software function.

number of memory ports.

An important challenge in hardware accelerator systems is to get the data to the computing core that needs it. Hiding data communication delay is needed to improve performance of the systems. In order to do this effectively, the resource allocation decision requires detailed and accurate information on the amount of data that is needed as input, and what will be produced as output. Evidently, there are dependencies among computations, since data produced by one kernel may be needed by another kernel. In order to have an efficient allocation scheme where the communication delays can be hidden as much as possible, a detailed profile of the data communication patterns is necessary for which the most appropriate interconnect infrastructure can be generated. Such communication patterns can be specific for each application and could lead to different types of interconnect. In this dissertation, we address the problem of automated generation of an optimized hybrid interconnect for a specific application.

1.2. DISSERTATION CHALLENGES

In state-of-the-art execution models of hardware accelerator systems in the literature, data input required for kernel computation is fetched to its local memory (buffers) when the kernel is invoked as described in [Cong and Zou, 2009] and [Canis et al., 2013]. This delays the start-up of kernel calculations until the whole data is available. Although there are some specific solutions to improve this communication behavior (presented in Section 2.4), those solutions are ad-hoc approaches for specific architectures or specific platforms. Moreover, those approaches have not taken the data communication pattern of the application into consideration. In contrast, we aim to provide a more generic solution and take the data communication pattern of the application into account.

In this work, we are targeting a generic heterogeneous hardware accelerator system containing general purpose processors and hardware accelerator kernels. The hardware accelerator kernels can be implemented by hardware fabrics such as FPGA, ASIC and GPU, etc. However, GPU interconnect is not reconfigurable in current day technology. Therefore, our discussion is mainly based on reconfigurable computing platforms.

Data communication in a hardware accelerator system can be optimized at both software and hardware levels (presented in Section 2.4). In this thesis we focus on the hardware level optimization. We therefore explore the following research questions:

Question 1 *How can data produced by an accelerator kernel be transferred to the consuming kernels as soon as it becomes available in order to reduce the delay of kernel calculation?*

As we presented above, most hardware accelerator systems transfer input data required for kernel computation to the local memory of the kernel whenever it is invoked and copy back output data when it is finished. This forces the kernel computing to wait for data movement to complete. In this work, we try to answer this question using a generic approach to improve the system performance.

Question 2 *Does it pay off to build a dedicated and hybrid interconnect that provides the most appropriate support for the communication patterns inside an application?*

Interconnect plays an important role in a multicore system. It not only contributes to system performance but also incurs hardware overhead. Therefore, we try to define a dedicated and hybrid interconnect that takes the data communication patterns inside an application into account; and try to see how efficient the hybrid interconnect is when compared to standard interconnect.

Question 3 *How can we achieve the most optimized system performance while keeping the hardware resource usage for the hybrid interconnect minimal?*

Building a hybrid interconnect that takes the communication patterns of an application into consideration to improve the system performance while keeping the hardware resource usage minimal is one of the main criteria. The reason for this requirement is that the more hardware resources are used, the more challenges are faced, such as power consumption or thermal emission. Therefore, we try to answer this question to achieve an optimized hybrid interconnect in term of system performance and hardware resource usage.

Question 4 *Can the reduction of energy consumption achieved by system performance improvement compensate for the increased energy consumption caused by more hardware resource usage for the hybrid interconnect?*

A multicore system has a defined energy budget. Designing a new hybrid interconnect to improve system performance can lead to an increase in power consumption due to more hardware resource required for the interconnect. This, in turn, will lead to increasing overall energy consumption. Therefore, we try to answer this question to clarify the power utilization of the hybrid interconnect.

Question 5 *Is the hybrid interconnect able to produce system performance improvement in both embedded and high performance computing systems?*

Embedded and high performance computing accelerator systems are different. While most embedded accelerator platforms implement both the host and the accelerator kernels on the same chip, high performance computing platforms build them on different chips. The host processor in high performance computing platform usually works at a much higher frequency than the host in the embedded computing platform. Moreover, the communication infrastructure bandwidth in the high performance computing platforms is larger than in the embedded ones. Therefore, we explore whether the hybrid interconnect pays off in both types of systems or not.

1.3. CONTRIBUTIONS

Based on the research questions presented in the previous section, we have been working on the interconnect of the multicore architecture, especially hardware accelerator systems, to solve those research challenges. The main contributions of the dissertation can be summarized as follows:

- We introduce an efficient execution model for a heterogeneous hardware accelerator system.

Based on a detailed and quantitative data communication profiling, a kernel knows exactly which kernels consume its output. Therefore, it can deliver the output directly to the consuming kernels rather than sending it back to the host. Consequently, this reduces the delay of the start-up of kernel calculation. This delivery process is supported by the hybrid interconnect dedicated for each application. The transfer process can be done in parallel with kernel execution.

- We propose a heuristic communication-aware approach to design a hardware accelerator system with a custom interconnect.

Given the fact that many hardware accelerator systems are implemented using embedded platforms where the hardware resource is limited, embedded hardware accelerator systems usually use a bus as the communication infrastructure. Therefore, we propose a heuristic approach that takes the data communication pattern inside an application into account to design a hardware accelerator system with an optimized custom interconnect. The approach is mainly useful for

embedded systems. A number of solutions are considered consisting of crossbar-based shared local memory, direct memory access (DMA), local buffer, and hardware duplication. An analytical model to predict system performance improvement is also introduced.

- We propose an automated approach using a detailed and quantitative communication profiling information to define a hybrid interconnect for each specific application, resulting in the most optimized performance with a low hardware resource usage and energy consumption.

Evidently, kernels and their communication behaviors are different from one application to the other. Therefore, a specific application should have a specific hybrid interconnect to get data efficiently to the kernels that need it. We call it hybrid interconnect as ultimately the entire interconnect will consist of not only a NoC but also uni- or bidirectional communication channels or locally shared buffers for data exchange. Although in our current experiments we statically define the hybrid interconnect for each application, the ultimate goal is to have a dynamically changing infrastructure in function of the specific communication needs of the application. The design approach results in an optimized hybrid interconnect while keeping the hardware resources usage for the interconnect minimal.

- We demonstrate our proposed hybrid interconnect in both an embedded platform and a high performance computing platform to verify the benefit of the hybrid interconnect.

Two heterogeneous multicore platforms are used to validate our automated hybrid interconnect design approach and the proposed execution model. Those are the Molen architecture implemented on a Xilinx ML510 board [Xilinx, 2009] and the Convey high performance computing system [Convey Computer, 2012]. Experimental results in both these platforms show the benefits of the hybrid interconnect in terms of system performance and energy consumption compared to the systems without our hybrid interconnect.

1.4. DISSERTATION ORGANIZATION

The work in this dissertation is organized in 8 chapters. Chapter 2 gives a summary on standard on-chip interconnect techniques in the literature and analyzes

their advantages and disadvantages. Many taxonomies to classify the on-chip interconnects are presented. A survey on the hybrid interconnect architectures in the literature is also shown. This chapter also presents the state-of-the-art hardware accelerator systems and we zoom in on their interconnect aspects. Data communication optimization techniques in the literature for such systems are also summarized in the chapter.

Chapter 3 discusses an overview of our approach to design a hybrid interconnect for a specific application using quantitative data communication profiling information. The data communication-driven quantitative execution model is also presented. To further improve the system performance, parallelizing kernel processing is also analyzed in this chapter.

Chapter 4 analyzes different alternative interconnect solutions to improve the system performance of a bus-based hardware accelerator system. A number of solution are presented: DMA, crossbar, NoC, as well as combinations of these. This chapter also proposes the analytical models to predict the performance for these solutions and implements them in practice. We profile the application to extract the data input for the analytical models.

Chapter 5 presents a heuristic-based approach to design an application specific hardware accelerator system with a custom[2] interconnect using quantitative data communication profiling information. A number of solutions are considered in this chapter. Those are crossbar-based shared local memory, DMA support parallel processing, local buffer, and hardware duplication. Experimental results with different applications are done to validate the proposed heuristic approach. We also analyze the contribution of each solution to system performance improvement.

Chapter 6 introduces an automated interconnect design strategy to create an efficient custom interconnect for kernels in a hardware accelerator system to accelerate their communication behavior. Our custom interconnect includes a NoC, shared local memory solution, or both. Depending on the quantitative communication profiling of the application, the interconnect is built using our proposed custom interconnect design algorithm. An adaptive data communication-based mapping for the hardware accelerators is proposed to obtain a low overhead and latency interconnect. Experiments on both an embedded platform and a high performance computing platform are performed to validate the

[2] In this work, we use two terminology *hybrid interconnect* and *custom interconnect* interchangeably.

proposed design strategy.

In Chapter 7, we present a case study of a heterogeneous hardware accelerator architecture to support streaming image processing. Each image in a dataset is preprocessed on a host processor and sent to hardware kernels. The host processor and the hardware kernels process a stream of images in parallel. The Convey hybrid computing system is used to develop our proposed architecture. The Canny edge detection application is used as our case study.

Finally, we summarize the list of our contribution and conclude this dissertation in Chapter 8. We also propose open questions and future research in this chapter.

2

Background and Related Work

In this chapter, we give a summary of state-of-the-art standard on-chip interconnects. Many taxonomies to classify the on-chip interconnects are presented. A survey on the hybrid interconnect architectures is discussed. Hardware accelerator systems in the literature are also presented where we zoom in on their communication infrastructures. We also give an overview on the data communication optimization techniques in the literature for hardware accelerator systems.

2.1. On-chip Interconnect

In modern digital systems, particularly in multicore systems, processing elements (PEs) are not isolated. They cooperate to process data. Therefore, the interconnection network (communication infrastructure) plays an important role to exchange data among the PEs as well as between the PEs and the memory modules. Choosing a suitable interconnection network has a strong impact on system performance. There are three main factors affecting the choice of an appropriate interconnection network for an underlying system. Those are performance, scalability and cost [Duato et al., 2002].

Interconnection networks connect components at different levels. Therefore, they can be classified into different groups [Dubois et al., 2014].

- On-chip interconnects connect PEs together and PEs to memory modules.

- I/O interconnects connect various I/O devices to the system communication infrastructure.
- Inter-system interconnects connect separated systems together. They include system area networks (SANs - connecting systems at a very short distances), local area networks (LANs - connecting systems within an organization or a building) and wide area networks (WANs - connecting multiple LANs at long distances).
- Internet is also a global and worldwide interconnect.

As a subset of a broader class - the interconnection network, on-chip interconnect transfers data between communicating nodes[1] in a system-on-chip (SoC). During the last decades, many on-chip interconnects have been proposed, along with the rising number of PEs in the systems. Figure 2.1 (adapted from [Matos et al., 2013]) summarizes the evolution of on-chip interconnects.

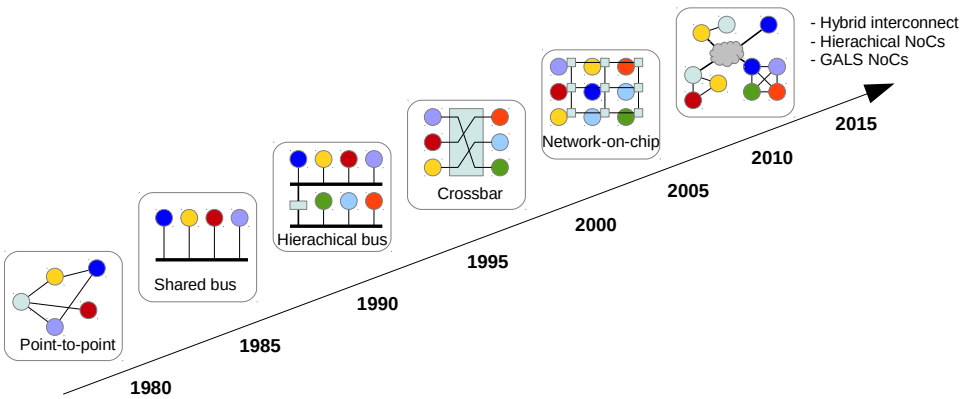


Figure 2.1: The evolution of the on-chip interconnects

There are many different ways to classify on-chip interconnects. Here, we list the five different well-known taxonomies.

Taxonomy 1 *Mechanism-based classification.*

Based on the mechanism upon which the processing elements communicate together, on-chip interconnects can be divided into two groups: shared memory and message passing [Pham et al., 2011].

[1] a node is any component that connects to the network such as a processing element or a memory module

- *Shared memory*: the idea of shared memory is that the system consists of shared memories that are accessed by the communicating processing elements. The producing PEs write data to the shared memory modules while the consuming PEs read data from those shared memories. Examples of this interconnect type are bus systems, directly shared local memory, and crossbars.

- *Message passing*: in this interconnect type, communication among PEs is carried out by explicit messages. Data from the source PE is encoded to interconnect packets and sent to the destination PEs through the interconnect. Examples of this interconnect type are Network-on-Chips (NoCs).

Taxonomy 2 *Connection-based classification.*

Based on the connection of the PEs, interconnects can be categorized into four major classes: shared medium networks, direct networks, indirect networks and hybrid networks [Duato et al., 2002].

- *Shared medium networks*: in this type, the transmission medium is shared by all the communicating nodes. Examples for this type of interconnect are buses, and directly shared local memory.

- *Direct networks*: in this scheme, each communicating node has a router, and there are point-to-point links to connect one communicating node to a subset of other communicating nodes in the network. Examples of this category are NoCs.

- *Indirect networks*: networks belonging to this category have nodes connected together by one or more switches. Examples of this interconnect types are crossbars.

- *Hybrid network*: in general, the hybrid networks combine shared medium and direct or indirect networks to alleviate the disadvantages of one type by the advantages of the other type such as increasing bandwidth with respect to shared medium networks and decreasing the distance between nodes in direct and indirect networks.

Taxonomy 3 *Communication link-based classification.*

Based on how to connect a PE and a memory module to other PEs and memory modules, interconnects can be categorized into two categories: static and dynamic networks [Grama et al., 2002].

- *Static networks*: A static network consists of dedicated communication links established among the communicating nodes to form a fixed network. Examples of this type of networks are NoCs and directly shared local memory.

- *Dynamic networks*: A dynamic network consists of switches and communication links. The links are connected together dynamically through the switches to establish paths among communicating nodes. Examples for this type of networks are buses and crossbars.

Taxonomy 4 *Switching technique-based classification.*

Based on the switching techniques, the mechanisms for forwarding message from the source nodes to the destination nodes, of the interconnects, they can be classified into two classes: circuit switching and packet switching [El-Rewini and Abd-El-Barr, 2005].

- *Circuit switching networks*: In this group of networks, a physical path is established between the source and the destination before data is transmitted through the network. This established path exists during the whole data communication period; no other source and destination pair can share this path. Examples of this interconnect network group are buses, crossbar, and directly shared local memory.

- *Packet switching networks*: The networks in this group partition communication data into small fixed-length packets. Each packet is individually transferred from the source to the destination through the network. Examples of this group are NoCs, which may use either wormhole or virtual cut-through switching mechanisms.

Taxonomy 5 *Architecture-based classification.*

Based on the interconnect architecture, interconnects can be classified into many different groups [Gebali, 2011; Kogel et al., 2006]. Here, we list only four well-known interconnects that are widely used in most hardware accelerator systems. Those are: directly shared local memory, bus, crossbar, and NoC.

- *Directly shared local memory*: In this interconnect scheme, PEs connect directly to memory modules through the memory ports as illustrated in Figure 2.2(a). Communication among the PEs is carried out through read and write operations.

- *Bus*: The bus is the simplest and most well-known interconnect. All the communicating nodes are connected to the bus as shown in Figure 2.2(b). Communication among the nodes follows a bus-protocol [Pasricha and Dutt, 2008].

- *Crossbar*: A crossbar is defined as a switch with n inputs and m outputs. Figure 2.2(c) depicts a 2×2 crossbar. A crossbar can connect any input to any free output. It is usually used to establish an interconnect for n processors and m memory modules.

- *NoC*: A NoC consists of routers or switches connected together by links. The connection pattern of these routers or switches forms a network topology. Examples of well-known network topologies are ring, 2D-mesh, torus or tree. Figure 2.2(d) illustrates a 2D-mesh NoC.

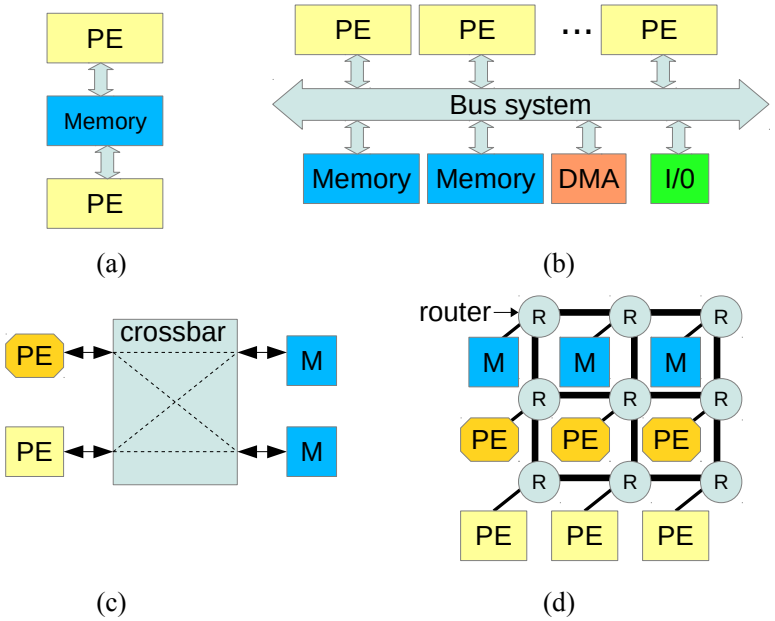


Figure 2.2: (a) Directly shared local memory; (b) Bus; (c) Crossbar; (d) Network-on-Chip

Table 2.1: Interconnect classifications overview

Taxonomy	DSLM[a]	Bus	Crossbar	NoC
Taxonomy 1	shared memory	shared memory	shared memory	message passing
Taxonomy 2	shared medium	shared medium	indirect network	direct network
Taxonomy 3	dynamic network	dynamic network	dynamic network	static network
Taxonomy 4	circuit switching	circuit switching	circuit switching	packet switching

[a]: Directly Shared Local Memory

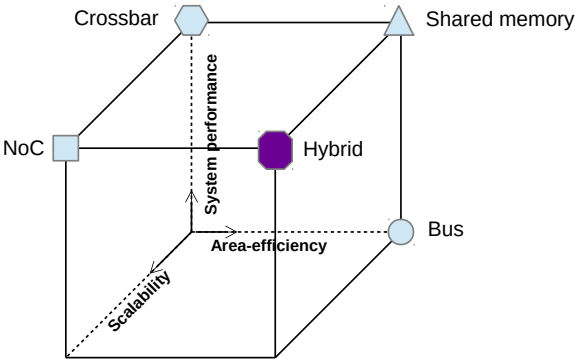


Figure 2.3: Interconnects comparison.

Table 2.1 shows the relationship between the taxonomies. Figure 2.3 illustrates the advantages and disadvantages of different interconnect types. While buses are simple and area-efficient, they suffer from low performance and scalability problems compared to the others because of the serialized communication [Sanchez et al., 2010]. A crossbar outperforms a bus in term of system performance because it offers separate paths from sources to destinations [Hur, 2011]. However, it has limited scalability because the area cost increases quadratically when the number of ports increases. While shared local memory can offer an area-efficient solution, its scalability is limited by the finite number of memory ports. Although NoCs have their certain advantages such as high performance and scalability, they suffer from a high area cost [Guerrier and Greiner, 2000]. Therefore, a hybrid interconnect with high performance, area-efficiency and high scalability is an essential demand.

2.2. SYSTEM-LEVEL HYBRID INTERCONNECT

In this section, we review proposed hybrid interconnects in the literature. In the previous section, we introduced five different taxonomies to classify the interconnects. Each interconnect group has its own advantages and disadvantages. For example, compared to the indirect interconnect group, direct interconnects are simpler in term of implementation but have lower performance while indirect interconnects provide better scalability but are accomplished with higher cost. Circuit switching interconnects are faster and have higher bandwidth than packet switching interconnects but they may block other messages because the physical path is reserved during the message communication. Meanwhile, many messages can be processed simultaneously in packet switching interconnects, however message partitioning produces some overhead. Therefore, in recent years, hybrid interconnects have been proposed to take the advantages of different interconnect types.

Hybrid interconnects can be classified into two groups. In the first group, a combination of different topologies of NoCs forms a hybrid interconnect, for example, a combination of a 2D-mesh topology and a ring topology. We name this group as *mixed topologies hybrid interconnect*. The second group includes hybrid interconnects that utilize multiple interconnect architectures, for example, a combination of a bus and a NoC. We name this group as *mixed architectures hybrid interconnect*. The following sections present the proposed hybrid interconnects of these groups.

2.2.1. MIXED TOPOLOGIES HYBRID INTERCONNECT

Network-on-chip topology [Jerger and Peh, 2009] refers to the structure upon which the nodes are connected together via the links. There are many standard topologies well presented in the literature. Figure 2.4 gives some examples of NoC topology including 2D-mesh, ring, hypercube, tree, and star. Although there are some certain advantages in those standard topologies, each topology suffers from some disadvantages, for example 2D-mesh has drawbacks in communication latency scalability, and the concentration of the traffic in the center of the mesh [Bourduas and Zilic, 2011] while ring topology does not offer a uniform latency for all nodes [Pham et al., 2011]. Therefore, hybrid topology or application-specific topology interconnects have been proposed. The following summary introduces some hybrid topology interconnects in the literature. The list is sorted by the publication year.

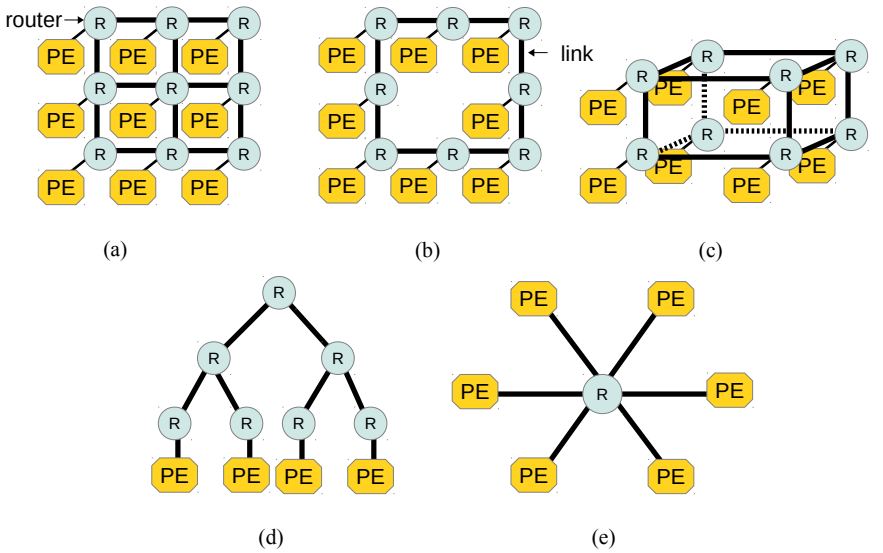


Figure 2.4: Examples of NoC topologies: (a) 2D-mesh; (b) ring; (c) hypercube; (d) tree; and (e) star.

CMesh (concentrated mesh) [Balfour and Dally, 2006] combines four communicating nodes into a group through a star connection. Those groups are connected together via a 2D-mesh network. Compared to the original mesh network, the CMesh network reduces the average hop count. As an extended CMesh network, the Flattened Butterfly network [Kim et al., 2007] adds dedicated links between the groups in a row or a column. With those point-to-point links, the maximum hop count of the Flattened Butterfly network is two. Simulation is used to evaluate both the network. The results show that CMesh has a 24% improvement in area-efficiency and a 48% reduction in energy consumption compared to other topologies. Compared to the mesh network, the Flattened Butterfly produces 4× area reduction while reducing 2.5× area when compared to CMesh.

Murali et al. [2006] proposed a design methodology that automated synthesizes a custom-tailored, application-specific NoC that satisfies the design objectives and the constraints of the targeted application domain. The main goal of the methodology is to design NoC topologies that satisfy two objective functions: minimizing network power consumption, and minimizing the hop-count. To achieve the goal, based on a task graph, the following steps are executed: 1) exploring several topologies with different number of switches; 2) automated performing floor-planning for the topologies; 3) choosing the topology that best op-

timizes the design objectives and satisfies all the constraints. Experimental results on an embedded platform using ARM processors as computing cores show that the synthesized topology improves system performance up to 1.73× and reduces the power consumption 2.78× in average when compared to the standard topologies.

The Mesh-of-Tree (MoT) interconnection network [Balkan et al., 2006] combines two sets of trees to connect processing elements (PEs) and memory modules. In contrast to other tree-based network architectures, where communicating nodes are connected to the leaf nodes, the communicating nodes are associated with the root nodes. The first set of trees, called the fan-out trees, is attached to PEs while the second set, called the fan-in set, is linked to memory modules. The leaf nodes of the fan-out set are associated with the leaf nodes of the fan-in set in an 1-to-1 mapping. The MoT network has two main properties: the path between each source and each destination is unique, and packets transferred between different sources and destinations will not interfere. Simulation is used to validate the proposed architecture. The results show that MoT can improve the network throughput by up to 76% and 28% when compared to butterfly and hypercube networks, respectively.

The hybrid MoT-BF network [Balkan et al., 2008] combining the MoT network and the area efficient butterfly network (BF) is an extended version of the MoT network. The main goal of this hybrid network is to reduce the area cost of the MoT network. Therefore, some intermediate nodes and leaf nodes of both the fan-in and fan-out trees are replaced by the 2×2 butterfly networks. The number of replaced intermediate nodes is the level of the MoT-h-BF network where h is the network level. Simulation is done to validate the architecture and compare the throughput with the previous version. According to the results, a 64 terminals MoT-BF reduces 34% area overhead with only 0.5% sacrificing throughput compared to the MoT network.

ReNoC [Stensgaard and Sparso, 2008] is a NoC architecture that enables the topology to be reconfigured based on the application task graph. In this work, each network node consists of a conventional NoC router wrapped by a topology switch. The topology switch can connect the NoC links to the router and the NoC links together (bypass the router). Therefore, different topologies can be formed based on the application task graph by configuring the topology switch. The final interconnect can be a combination of rings and meshes or even point-to-point links interconnect. The experimental results with the ASIC 90nm tech-

nology show that only 25% hardware resource is needed for the ReNoC compared to a static mesh while energy consumption is reduced by up to 56%.

G-Star/L-Hybrid [Kim and Hwang, 2008] is a hybrid interconnect including a star topology global network and mixed topology (star and mesh) local networks. The main purpose of this hybrid network is to reduce the packet drop rate. The author conducted many different topology combinations with some different applications and concluded that combining both the star and the mesh topology is the most optimized solution. Simulation results show that compared to other topologies, up to 45.5% packet drop was reduced by the proposed hybrid interconnect. Power consumption and area overhead are also better than other topologies.

VIP [Modarressi et al., 2010] is a hybrid network benefiting from the scalability and resource utilization advantages of NoCs and superior communication performance of point-to-point dedicated links. To build the hybrid interconnect, the following steps are done based on the application task graph: 1) physically map the tasks to different nodes of a 2D-mesh NoC; 2) construct the point-to-point links between the tasks as much as possible; 3) re-direct the flow for which messages are traveled following the point-to-point link in such a way that the power consumption and latency of the 2D-mesh NoC is minimized. A NoC simulator tool is used to evaluate the architecture. The experimental results show that VIPs reduce the total NoC power consumption by 20%, on average, over other NoCs.

Bourduas and Zilic [2011] proposed several hierarchical topologies that use the ring networks to reduce hop counts and latencies of global (long distance) traffic. In this approach, a mesh is partitioned into sub-meshes (a sub-mesh is the smallest mesh in the system, a 2×2 mesh). Four sub-meshes are connected together by a ring forming a local mesh. Consequently, local meshes are connected together by another ring. The ring-mesh bridge component is also designed for transferring packets between mesh nodes and ring nodes. Moreover, two ring architectures are also implemented. The first is a slotted simplicity and low-cost ring architecture while the second uses wormhole routing and virtual channel that provide flexibility and best performance. Simulation validated the claims of the proposed architecture. The results show that the proposed hybrid topologies outperform the mesh network when the number of nodes is smaller than 44.

DMesh [Wang et al., 2011] composes of two sub-networks called E-subnet

and W-subnet in which each router is added diagonal links to neighbor routers. E-subnet is responsible for transferring eastward packets while W-subnet is responsible for westward traffic. Each router consists of two sub-routers, one E-router for E-subnet and one W-router for W-subnet. When the source PE starts a message transmission, packets are injected into the network via either E-router or W-router depending on the direction of the destination PE. New routing algorithm is also proposed for the architecture. A SystemC-based simulator tool is used to evaluate the proposed network. The results show that DMesh outperforms the compared network in an 8×8 network.

PC-Mesh [Camacho et al., 2011] is another extended version of the CMesh network. The PC-Mesh network uses some other 2D-mesh networks to connect groups of four adjacent nodes which are not grouped in the original CMesh network yet. The benefits of the PC-Mesh network are its fault tolerance degree and the lower latency in terms of hops. Because one node is connected to more than one switches, an injection algorithm is proposed to adapt the utilization of the added 2D-mesh networks to the current injection load of the node. Simulation is used to validate the proposed architecture. The results show that PC-Mesh can reduce execution time by a factor of 2 and energy consumption by 50% when compared to CMesh.

Yin et al. [2014] proposed a hybrid-switch NoC that combines point-to-point links and a standard 2D-mesh NoC. The dedicated point-to-point links are established between the frequently communicating nodes by explicit configuration messages. In another point of view, the architecture supports both packet and circuit switching in which packet-switched messages are buffered, routed and then forwarded at each router; while circuit-switched messages follow dedicated links without incurring additional buffering/routing overhead. Simulation results show that the proposed direct links can improve system performance by up to 12% and reduce energy consumption by up to 24% when compared to the original NoC.

Swaminathan et al. [2014] proposed a hybrid NoC topology that combines triple topologies: 2D-mesh, torus and folded. The mesh links connect two adjacent routers while the folded-like links bridge the odd routers in a row or a column together as well as the even routers in a row or a column together. The torus-like links connect two routers at the boundary of a row or a column. The new routing algorithm for the hybrid NoC topology is proposed in this work. Due to the combination of triple topologies, the hybrid NoC topology reduces the av-

erage hop count compared to the original topologies and improves the throughput. Simulation is used to evaluate the proposed architecture. The results show that the proposed hybrid NoC can improve system performance by up to 26% when compared to mesh network.

Table 2.2 summarizes all the proposed mixed topologies hybrid interconnects already presented. As shown in the table, most of the proposed mixed topologies hybrid interconnects do not take specific application parameters, such as data communication pattern, into consideration. Two of them use task graphs to design the hybrid NoCs. However, the task graph does not show actual communication pattern inside an application. One approach uses communication rate to establish the links. However, communication rate may change during the execution time.

2.2.2. MIXED ARCHITECTURES HYBRID INTERCONNECT

Although directly shared local memory, bus, crossbar, and NoC are used in most computing systems, they suffer from their own disadvantages as already analyzed in Section 2.1. Therefore, many studies in the literature have proposed hybrid interconnects that combine one or more interconnect types together to compensate disadvantages of one type by advantages of other types. In this section, we summarize mixed architectures hybrid interconnects in the literature. The list is sorted by the publication year.

dTDMA/NoC [Richardson et al., 2006] hybrid interconnect is composed of buses and a NoC. A bus is used to connect a number of frequently communicating PEs belonging to an affinity group while communication between PEs not belonging to an affinity group is accomplished by the NoC. This proposed hybrid architecture is based on two heuristics: 1) Buses provide a better performance than NoCs for a group of 9 PEs or fewer; 2) NoC performance degrades much faster than bus performance with increasing load rate. Therefore, the PEs are grouped into affinity groups based on their frequently communicating behavior. All PEs in one affinity group are linked by a bus. Each affinity group is associated with one NoC router through a bridge. All PEs that are not assigned into any affinity group are also connected to NoC routers. Simulation results show that the hybrid interconnect outperforms the original NoC in both performance and energy consumption. The worst-case latency reduction and power consumption reduction are 15.2% and 8%, respectively, when compared to mesh network.

MECS [Grot et al., 2009] (Multidrop Express Channels) is a hybrid intercon-

Table 2.2: Mixed topology hybrid interconnect summary

Proposal	**Combined topologies**	**Input data**[a]	**Experimental platform**	**Year**
CMesh	Mesh/Star	Static[b]	Simulation	2006
Murali et al.	Various[c]	User constraints	Embedded platform	2006
MoT	Mesh/Tree	Static	Simulation	2006
Flattened Butterfly	Mesh/Star/ P2P[d]	Static	Simulation	2007
MoT-BF	Mesh/Tree/ Butterfly	Static	Simulation	2008
ReNoC	Various	Task graph	Embedded platform	2008
G-Star/L-Hybrid	Mesh/Star	Static	Simulation	2008
VIP	Mesh/P2P	Task graph	Simulation	2010
Bourduas et al.	Mesh/Ring	Static	Simulation	2011
DMesh	Mesh/Mesh	Static	Simulation	2011
PC-Mesh	Several Meshes	Static	Simulation	2011
Yin et al.	Mesh/P2P	Communication rate	Simulation	2014
Swaminathan et al.	Folded/Mesh/ Torus	Static	Simulation	2014

[a] Which input data the proposal uses to design the proposed architecture, for example task graph or communication pattern.
[b] *Static* means that the proposal does not use any input data from any application/domain.
[c] *Various* means that many topologies can be used depending on the application.
[d] Point-to-point.

nect that combines a CMesh NoC [Balfour and Dally, 2006] and bus-like one-to-many channels. The bus-like one-to-may channel is similar in architecture as a bus, but only the master node can send data to one or many slave nodes connected to the channel. Each CMesh's group connects with $2(n-1)$ channels in which each channel connects all groups within a row or a column together. Because of the one-to-many channels, multicast and broadcast are supported with a little additional cost. Simulation with both synthetic and application-based workload shows the benefits of the interconnect compared to the CMesh and the Flatten Butterfly [Kim et al., 2007] interconnects. In a 64-terminal network, MECS offers a 9% latency advantage when compared to other topologies.

BENoC (Bus-enhanced NoC) [Manevich et al., 2009] is a hybrid interconnect in which a NoC is equipped with a specialized bus. The bus has low and predictable latency. The bus is used for system-wide distribution of control signals as well as performs broadcast and multicast. Therefore, the complexity and cost of broadcast operations in the NoC can be avoided by using the bus, because broadcast usually transmits short messages. Simulation results show that the BENoC provides an execution speedup around 3× on average compared to a classic NoC.

Das et al. [2009] proposed a hierarchical hybrid on-chip interconnect that uses both buses and a NoC. Eight PEs are connected by a bus to form a local network. Each bus is associated with a router of a 2D-mesh NoC through a bus interface to form the global network. A network transaction in the hybrid network can be either entirely carried out by the bus or will incur global transactions and an additional local transaction in order to reach the destination. Simulation with synthetic benchmark is done to evaluate the proposed architecture. The results show that the proposed hybrid interconnect improves system performance by up to 14% compared to mesh network.

RAMS [Avakian et al., 2010] is a reconfigurable hybrid interconnect that consists of bus-based subsystems connected through routers forming a mesh NoC. Based on the conclusion that when the number of PEs is small (vary between 1 and 8, depending on applications), bus-based systems outperform NoC-based systems; RAMS has scalable bus-based multiprocessor subsystems on each node in the NoC. PEs are attached to bus segments. Bus segments are connected together through switches. Based on the memory access rate, the operation system configures the switches to form bus-based subsystems. NoC simulator tool is used to compare the proposed RAMS interconnect to 2D-mesh NoC. The results

show that RAMS outperforms the original NoC in term of system performance.

Tsai et al. [2010] proposed a hybrid interconnect that consists of a NoC and buses. The NoC connects bus-based subsystems and single IP cores through their routers. Based on the communication graph that shows bandwidth requirements between application's functions, the approach classifies the IP cores implementing the functions into to affinity groups. IP cores that cannot be grouped are used as single IP cores. IP cores in an affinity group are connected together through a bus to form a subsystem. The subsystems are attached to the routers via bridges. Simulation is used to evaluate the proposed hybrid interconnect. The results show that up to 17.6% latency reduction was obtained.

HNoC [Zarkesh-Ha et al., 2010] is a hybrid interconnect with local buses and a global 2D-mesh. The HNoC hybrid interconnect uses local buses for nearest-neighbor communication and a 2D-mesh NoC for global interconnect. In other words, besides the 2D-mesh NoC, every two PEs that connect to two adjacent routers are linked by a bus. Those buses perform all the nearest-neighbor traffic. Therefore, traffic on the global network is reduced, which results in increased throughput and reduced energy consumption. HNoC is implemented in a system simulator to verify and evaluate. The experimental results show that HNoC improves throughput by 4.5× and reduces energy consumption by 58% when compared to a conventional NoC topology.

Giefers and Platzner [2010] proposed a hybrid interconnect that contains triple architectures: a reconfigurable mesh that can be configured as buses, a classical NoC, and a barrier network. The reconfigurable mesh consists of switches connected to PEs. PEs have control over a local switch and can dynamically reconfigure the switch. The PEs are also connected to the NoC routers. The barrier network is used to manage the synchronization of the PEs. An FPGA-based multicore prototype is used to validate and evaluate the hybrid interconnect. Experiments with the Jacobi algorithm shows that combination of the three networks provides the highest performance.

MORPHEUS [Grasset et al., 2011] is a heterogeneous accelerator system that uses three different components as the system interconnect. The host processor uses a control bus to handle control, synchronization, and debug all the resources. Configuration bitstreams for the hardware accelerators are transferred through another bus called configuration bus. Application data is transferred by a high-throughput NoC-based interconnect structure that allows direct access to the external Flash/SDRAM/DDRAM. A chip prototype is built to test the system

as well as the interconnect.

The *duo* [Jin et al., 2012] hybrid interconnect consists of a baseline 2D-mesh NoC augmented with a bus-like reconfigurable multidrop channels (the same in the MECS [Grot et al., 2009] hybrid interconnect). Due to the reconfiguration ability, each row or each column has only one channel instead of $2(n-1)$ in MECS. Communication traffics of applications are traced, and the applications are classified into classes based on their communication behavior. The channels are configured for each application class using the communication behavior of the class. Simulation is used to evaluate the proposed hybrid interconnect. The results show that network latency and energy consumption can be reduced on average by 15% and 27%, respectively when compared to mesh network.

Zhao et al. [2012] proposed a hybrid interconnect in which buses can share links with a NoC to form a bus-NoC hybrid interconnect. In order to share the links between buses and a NoC, a bus-switch component is added to each router. The component is a programmable switch that can connect segments (NoC links) to form a bus. When the links form the bus, they do not connect to the routers. When links are used as part of the bus, packets need to stay inside the router until the bus transaction is over. The hybrid interconnect is evaluated by a simulator to compare with the baseline NoC. The results show that the proposed hybrid interconnect improves system performance by up to 12% and saves energy consumption by up to 37% compared to the baseline network.

Todorov et al. [2014] proposed a deterministic synthesis approach to design a hybrid application-specific interconnect that contains buses and NoC routers. Input of the synthesis approach is the use-cases with bandwidth required, latency constraints, and packet size distribution information. Based on those use-cases, PEs are partitioned into clusters. Clusters with low communication bandwidth are connected to the shared buses while the other clusters are attached to the routers. The shared buses are linked to the routers through the network interfaces. A deadlock free flow routing algorithm is also proposed for the hybrid interconnect. Simulation results show that the proposed interconnect has almost the same latency compared to the classic NoC while reducing the hardware cost by up to 22.6% compared to conventional NoC topologies.

Table 2.3 summarizes all the proposed mixed architectures hybrid interconnects. As shown in the table, all the mixed architectures hybrid interconnects combine buses and a NoC to form hybrid interconnects. Beside the static designs, communication rate is usually used as input data to design the hybrid in-

Table 2.3: Mixed architecture hybrid interconnect summary

Proposal	Combined architectures	Input data[a]	Experimental platform	Year
dTDMA/NoC	Bus/NoC	Communication rate	Simulation	2006
MECS	Bus-like/NoC	Static	Simulation	2006
BENoC	Bus/NoC	Static[b]	Simulation	2009
Das et al.	Bus/NoC	Static	Simulation	2009
RAMS	Bus/NoC	Memory access rate	Simulation	2010
Tsai et al.	Bus/NoC	Communication bandwidth	Simulation	2010
HNoC	Bus/NoC	Static	Simulation	2010
Giefers et al.	Bus/NoC/ Barrier	Static	FPGA-based platform	2010
MORPHEUS	Bus/NoC	Static	ASIC-based platform	2011
duo	Bus-like/NoC	Communication rate	Simulation	2012
Zhao et al.	Bus/NoC	Static	Simulation	2012
Todorov et al.	Bus/NoC routers	Bandwidth and Latency constraints	Simulation	2014

[a] Which input data the proposal uses to design the proposed architecture, for example task graph or communication pattern.
[b] *Static* means that the proposal does not use any data from any application/domain.

terconnect. However, communication rate may change from time to time. None of the above proposed hybrid interconnects take application quantitative data communication pattern into account.

2.3. INTERCONNECT IN HARDWARE ACCELERATOR SYSTEMS

In recent years, many hardware accelerator systems have been proposed for general purpose computing as well as for specific applications (domains). Figure 2.5 presents a generic architecture of hardware accelerator system. In such system, the host processor can be a general high-performance CPU (e.g., x86 Intel CPU)

or an embedded processor (e.g., Xilinx PowerPC) or a soft processor (e.g., MicroBlaze or Nios). The kernels are implemented in hardware fabric such as FPGA, DSP, GPU, etc. While the host processor uses the main memory to store application data, the kernels have their local memories to store local data (data cache) to improve the parallelism between the kernels. A kernel communicates with the host, other kernels and I/Os through a communication infrastructure.

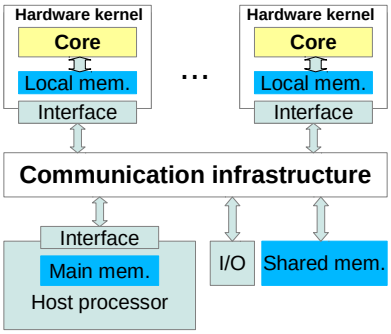


Figure 2.5: A generic hardware accelerator architecture

The following review classifies the hardware accelerator systems presented in the literature into four different groups based on the communication infrastructure (the interconnect) of the system.

- *Bus-based interconnect:* Molen [Vassiliadis et al., 2004], Warp processor [Lysecky and Vahid, 2009], IMORC [Schumacher et al., 2012], the target systems in [Canis et al., 2013; Ismail and Shannon, 2011; Pilato et al., 2012], and IBM's PowerEN [Heil et al., 2014] use a bus as the communication infrastructure. In these systems, data is transferred between the main memory and the local memories of the kernels through the bus. The host delivers data input to the kernel when it is invoked and collects the result when the kernel finishes computation through the bus. Other modules such as I/O, DMA, interrupt controller, etc. are also connected with the bus.

- *NoC-based interconnect:* The MORPHEUS system [Grasset et al., 2011; Voros et al., 2013] uses the Spidergon NoC for data communication of kernels and memory modules.In the target system in [Chung et al., 2011], and [Chung et al., 2012], a CoRAM element in each kernel collects data input for the kernel from the memory modules and sends the result back to them through a NoC. The P2012 architecture [Benini et al., 2012] uses an asynchronous

NoC for communication among the kernels while communication between the host and the kernels is done by direct memory access (DMA).

- *Shared memory:* Shared memory is used in many commercial hardware accelerator systems for high performance computing. Intel proposes a system using a Front Side Bus (FSB) [Ling et al., 2009] to enable both the host and FPGA access the shared memory. Convey [Convey Computer, 2012] uses a Hybrid-core Globally Shared Memory (HGSM) controlled by Convey's Hybrid-core Memory Interconnect (HCMI) for communication between the host and the kernels as well as among the kernels. The IBM Power 8 uses IBM's Coherence Attach Processor Interface (CAPI) [Stuecheli, 2013] to allow coherent memory sharing between the host processor and FPGA. Microsoft introduces their hardware system, called Catapult [Putnam et al., 2014], for accelerating large-scale datacenter services in which shared memory technique is used for data communication between the host and the accelerators. Shared memory mechanism is also used in [Willenberg and Chow, 2013] through a remote memory access infrastructure.

- *Crossbar:* The research in [Betkaoui et al., 2011] proposed a framework for accelerating large graph problems. The target system includes graph processing elements (GPEs) connected with memory modules through a full crossbar. The crossbar contains three different components: FIFOs, an arbiter and multiplexer. The round-robin algorithm is used to schedule communication between the GPEs and the memory. The work in [Cong and Xiao, 2013] proposed an optimized crossbar with the fewest switches while keeping high routability between the accelerators and the shared memories. The crossbar can be reconfigured upon accelerator launch so that each memory module is connected to only one accelerator.

While the prototypes described in [Canis et al., 2013; Chung et al., 2012; Cong and Xiao, 2013; Ismail and Shannon, 2011; Lysecky and Vahid, 2009; Pilato et al., 2012; Vassiliadis et al., 2004; Willenberg and Chow, 2013] implement the host and the kernels on the same chip (embedded hardwired or soft processor as the host), the implementation of [Benini et al., 2012; Betkaoui et al., 2011; Convey Computer, 2012; Ling et al., 2009; Putnam et al., 2014; Schumacher et al., 2012; Stuecheli, 2013; Voros et al., 2013] uses different chips for the host and the kernels.

2.4. DATA COMMUNICATION OPTIMIZATION TECHNIQUE

Data communication of a hardware accelerator system can be optimized at two different levels: software and hardware. The following sections present the techniques in the literature to optimize data communication at those levels.

2.4.1. SOFTWARE LEVEL OPTIMIZATION

Software level optimization is a set of operations or services executed by the host to speed up the software-hardware intercommunication. Optimization at software level can be applied for an existing hardware accelerator system without any hardware modification. However, software optimizations depend on the communication behavior of the running application. Additionally, some software level optimizations require specific modules/functions supported by the hardware platform.

Transferring data input/output between the main memory and the local memory in parallel with the kernel computation is one of the optimizations for data communication reported in [Cong and Zou, 2009]. Research in [Ismail and Shannon, 2011] developed an operating system service to establish a direct memory map to the address space of kernels and to enable arrays of data to be copied in a single access. Curreri et al. [2012] proposed a visualization tool to detect data communication bottleneck in an FPGA-based accelerator system using DMA to transfer data from the host to the kernels or among the kernels. Based on the identified bottleneck, designers can increase or decrease the DMA block size and bandwidth to improve the system performance. Pouchet et al. [2013] proposed a framework to optimize data reuse for a class of programs through which the data communication overhead was reduced. In [Goringer et al., 2010], the authors present the partitioning of an application between several processing elements (SW/SW partitioning) at the function-level, as well as HW/SW partitioning utilizing some profiling information. The work in [Ashraf et al., 2012] extended the QUAD toolset [Ostadzadeh et al., 2012] to provide the information about the unique data values involved in inter-function data-communication. The authors have utilized this extended information to perform the HW/SW partition to optimize the data-communication.

2.4.2. HARDWARE LEVEL OPTIMIZATION

Hardware level optimization is a hardware implementation targeted to specific platforms to speed up the software-hardware or hardware-hardware intercom-

munication. In recent years, some efficient interconnect architectures dedicated for a hardware fabric have been proposed to exploit the advantages of the fabric such as DESA NoC [Roca et al., 2012] or low-cost and specific-application crossbar in [Hur et al., 2012; Murali et al., 2007] developed for FPGAs. Additionally, many hybrid interconnects both in mixed NoC topologies and in mixed architectures have been proposed to improve the hybrid interconnect throughput or reduce hardware cost as presented in Section 2.2.

Moreover, other optimization techniques have been proposed to accelerate the communication behavior of the kernels in a hardware accelerator system because data communication is usually a primary anticipated bottleneck for system performance. Research in [Choi et al., 2012] proposed a multi-ported cache design for communication of multiple accelerator kernels in an FPGA-based accelerator system. However, this proposal is system-dependent since they assume that the on-chip memory can work at 2× the speed of the system clock (clock for kernels). Another interesting work is the CoRAM architecture [Chung et al., 2011]. In this work, CoRAM modules are used to collect data input required for kernel computation efficiently and write back the result to the memory. A software control task is used to manage the execution of the CoRAM modules and inform the kernels when data input is ready. In the IMORC architecture [Schumacher et al., 2012], asynchronous FIFOs are inserted into the communication channels between the cores and the memories to provide a sufficient bandwidth as well as to help improving the system performance by decoupling core execution and memory access.

However, all the above approaches have not taken the actual data communication pattern among the kernels into consideration yet. The work described in this dissertation uses data communication profiling information of each application/domain to automatically define an efficient hybrid interconnect for the kernels.

3

COMMUNICATION DRIVEN HYBRID INTERCONNECT DESIGN

INTERCONNECT in a heterogeneous multicore system, particularly in a hardware accelerator system, plays an important role especially when the number of cores is rising. Each interconnect type presented in Chapter 2 has its own advantages and disadvantages. Moreover, communication patterns are different from application to application. A specific application should have a specific interconnect dedicated to its communication patterns. The specific interconnect should have an optimized performance while keeping hardware resource usage minimal. Therefore, in this chapter, we present an overview of our approach to design a hybrid interconnect for a specific application using quantitative data communication profiling information. The data communication driven quantitative execution model is also presented. Based on the data communication profiling information, the data output from one kernel is delivered to the consuming kernels in parallel with the kernel execution. Therefore, the consuming kernels do not need to collect these data input when they are invoked.

3.1. OVERVIEW HYBRID INTERCONNECT DESIGN

In this section, we define the terminology used in the dissertation and present an overview of our approach.

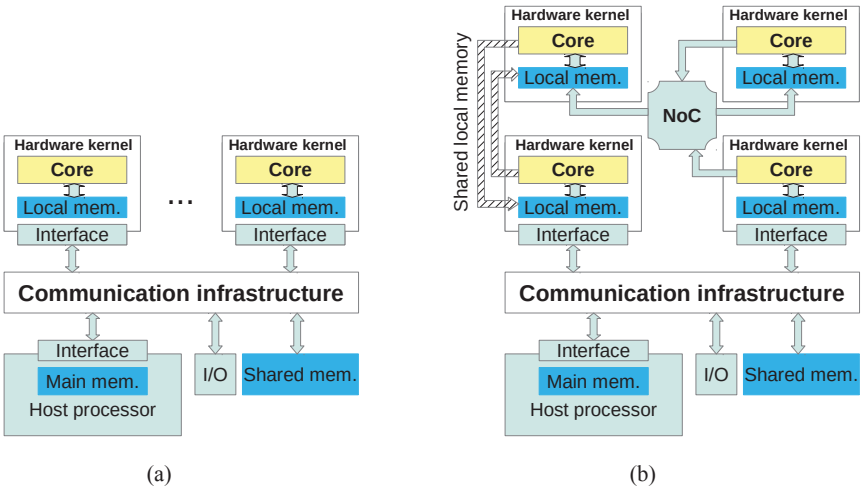


Figure 3.1: (a) The generic FPGA-based accelerator architecture; (b) The generic FPGA-based accelerator system with our hybrid interconnect.

3.1.1. TERMINOLOGY

In a generic hardware accelerator system (as depicted in Figure 3.1(a)), the *communication infrastructure* is a predefined system backbone upon which data is transferred between the host and the kernels as well as among the kernels. The communication infrastructure is different from system to system. It can be a bus, a NoC, shared memory, or a crossbar. In this work, we refer to the terminology communication infrastructure as the original interconnect of the system.

In this work, the *hybrid interconnect* terminology refers to our proposed infrastructure that is used for data communication among the hardware accelerator kernels to speed up system performance, which consists of different interconnect types such as NoC, shared local memory, or crossbar. The accelerator system using our approach includes both the original communication infrastructure to exchange parameters as well as data between the host and the kernels and the hybrid interconnect to transfer data from one kernel to the other kernels. Figure 3.1(b) illustrates the generic accelerator system with our hybrid interconnect.

3.1.2. OUR APPROACH

In this work, the detailed profile of the data communication pattern is used to define a hybrid interconnect which can lessen the data communication bottle-

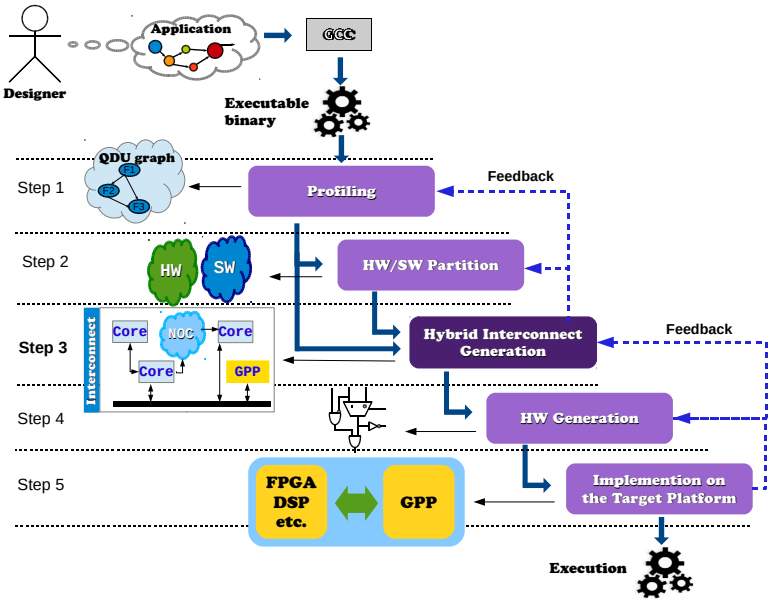


Figure 3.2: Hybrid interconnect design steps

neck issue and improve system performance. Based on the detailed profile, a kernel knows exactly which other kernels will consume its output. Therefore, the kernel can deliver its output directly to the consuming kernels as soon as output data is available. This delivery process is supported by the hybrid interconnect and can be done in parallel with kernel execution. In order to define the hybrid interconnect, the design steps as depicted in Figure 3.2 are used.

STEP 1: PROFILING

Profiling provides characteristics of an application such as execution time, communication patterns, etc., to drive the subsequent steps. In this work, the application is analyzed by the GNU profiler, gropf [Graham et al., 1982], to recognize the computationally intensive functions. Those functions are good candidates for hardware acceleration.

In order to generate the data communication profiling driving the hybrid interconnect design, the QUAD toolset [Ostadzadeh, 2012] is used. Conventional profilers such as gprof or Valgrind [Nethercote and Seward, 2007] do not separate actual computational time from data communication time coming from bus access, memory access, etc. QUAD provides a comprehensive overview of the memory access behavior of an application by a QDU (Quantitative Data Us-

age) graph output (an example of a QDU graph is shown in Figure 3.3). QUAD toolset is based on PIN *D*ynamic *B*inary *I*nstrumentation (DBI) framework [Luk et al., 2005]. QUAD traces each memory read and write access to record necessary information regarding the data communication among functions. When a function writes data to a memory location, it is marked as the producer of the data. The function reading data from the memory location is known as the consumer of this data. The QUAD tool reports an amount of data communication between a producer and a consumer in bytes. Memory addresses are also analyzed to calculate the number of unique locations used in the data communication. This information is reported in the end as Unique Memory Addresses (*UNMAs*). Furthermore, the number of bytes uniquely communicated is also reported as Unique Data Values (*UNDVs*).

STEP 2: HARDWARE-SOFTWARE PARTITION

The main purpose of the hardware-software partition is to decompose the application into parts that can be mapped to the host processor or to the hardware kernels. The main objective of this partition is to improve system performance, consequently, to achieve speed-up. In order to reach this goal, the computationally intensive parts are usually mapped to the hardware accelerators while the other parts are executed by the host.

STEP 3: HYBRID INTERCONNECT GENERATION

The main objective of this work is to define a hybrid interconnect to optimize the data communication behavior of the hardware kernels while keeping the hardware resource usage for the hybrid interconnect minimal. Based on the results of Steps 1 and 2, this step defines a dedicated hybrid interconnect for each application to help the kernels deliver their output to other kernels in an optimal way. The detail hybrid interconnect generation is explained in the later chapters.

STEP 4: HARDWARE GENERATION

In this step, the hardware implementation of the accelerated parts is generated. Although we target a generic hardware accelerator system in which FPGAs, ASICs, or GPUs can be used as the hardware accelerator fabric, GPU interconnect is not reconfigurable in current day technology. Therefore, we develop our experiments using reconfigurable computing platforms. High level synthesis tools such as Xilinx Vivado [Xilinx, 2014], DWARV [Nane et al., 2012], etc., generate the synthesizable hardware descriptions for the accelerated parts from the original application source code.

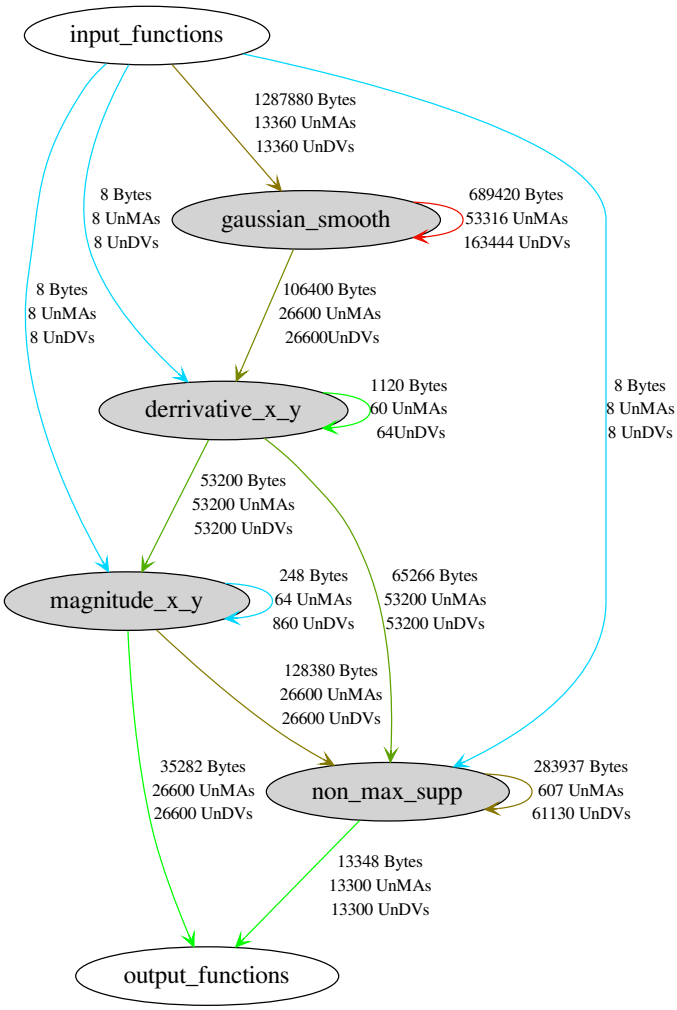


Figure 3.3: Example of a QDU graph

STEP 5: IMPLEMENTATION ON THE TARGET PLATFORM

The system is synthesized and implemented on target platforms using the supporting tools related to those platforms. Finally, the application is executed on those platforms by both the host and the hardware accelerator kernels. The host controls the execution of the kernels as well as data communication between the host and the kernels following the ideal execution model presented in the next section.

3.2. DATA COMMUNICATION DRIVEN QUANTITATIVE EXECUTION MODEL

This section presents a baseline execution model that we use to compare our proposed execution model. Our proposed execution model in which data input for a kernel is delivered to its local memory as soon as possible is presented. To further optimize the system performance, parallelizing kernel processing is also discussed.

3.2.1. BASELINE EXECUTION MODEL

Similar to most presented hardware accelerator systems, we use a baseline system that contains one host processor and some accelerator kernels. The host processes part of the application in software while the accelerator kernels process some computationally intensive functions of the application. The data input required for kernel computation is fetched to the local memory of the kernel, and the result is copied back to the host after the kernel finished. Data is transferred through the communication infrastructure.

Consider a hardware accelerator system with n kernels; a kernel i (with $0 \leq i \leq n-1$) is defined as in Equation 3.1

$$HW_i(\tau_i, D^H_{i(in)}, D^K_{i(in)}, D^H_{i(out)}, D^K_{i(out)}) \tag{3.1}$$

in which:

- τ_i is the computation time of the kernel;
- $D^H_{i(in)}$ and $D^K_{i(in)}$ are the total amount of data input for the kernel generated by the functions in the host and by the other kernels, respectively;
- $D^H_{i(out)}$ and $D^K_{i(out)}$ are the total amount of data output of the kernel consumed by the functions in the host and by the other kernels, respectively.

According to the baseline model, all data input ($D_{i(in)} = D^H_{i(in)} + D^K_{i(in)}$) required for the execution of kernel i is fetched from the main memory and all output result ($D_{i(out)} = D^H_{i(out)} + D^K_{i(out)}$) is copied back to the main memory. However, we distinguish data of the host ($D^H_{(in/out)}$) and of the other kernels ($D^K_{(in/out)}$) to make a comparison to our proposed model later on.

Following this model, the total baseline execution time (T_b) of the n kernels is shown in Equation 3.2, in which θ is the average time for transferring one byte of data through the communication infrastructure. This value is system-dependent. In this equation, $\sum_{i=0}^{n-1} \tau_i$ is the total computation time of all the kernels while $\sum_{i=0}^{n-1}(D_{i(in)} + D_{i(out)})\theta$ is the total communication time (transferring data). While computation time depends mainly on the algorithm itself, communication time depends on the data movement behavior.

$$T_b = \sum_{i=0}^{n-1} \tau_i + \sum_{i=0}^{n-1} (D_{i(in)} + D_{i(out)})\theta, \tag{3.2}$$

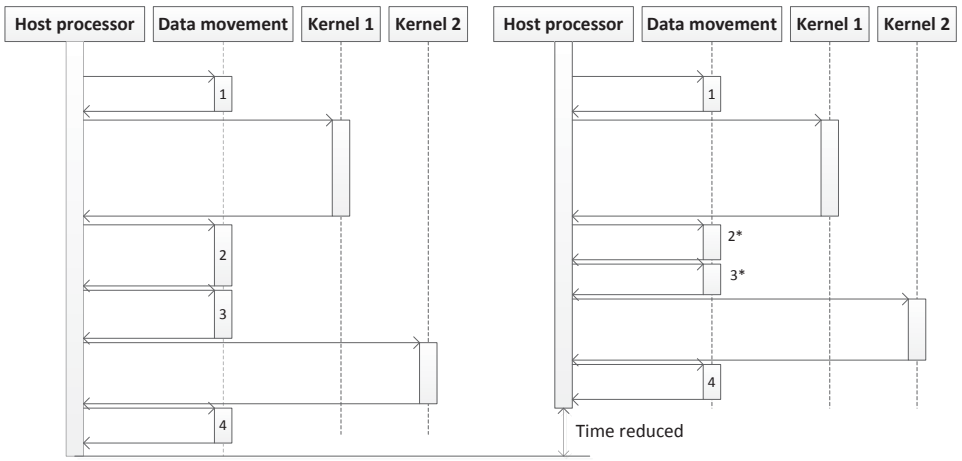
where $D_{i(in)} = D^H_{i(in)} + D^K_{i(in)}$ and $D_{i(out)} = D^H_{i(out)} + D^K_{i(out)}$.

Although the fetching phase can be done in pipeline with the computation phase, this mechanism depends on the actual behavior of the application. Therefore, we use a general model that is compatible with most applications (domains) as a baseline model.

3.2.2. IDEAL EXECUTION MODEL

Based on the data communication profiling, a kernel knows exactly which kernels will consume its output. The kernel can delivers it output directly to the consumed kernels through the hybrid interconnect. Consequently, only data input for the kernels generated by the functions on the host ($D^H_{i(in)}$) and data output of the kernels consumed by the functions on the host ($D^H_{i(out)}$) needs to be transferred through the communication infrastructure. Other data ($D^K_{i(in)}$ and $D^K_{i(out)}$) is transferred from kernel to kernel by the hybrid interconnect in parallel with the kernels' execution. Therefore, the data communication among the kernels is hidden. The total execution time of the n kernels following the ideal execution model (T_{ideal}) is shown in Equation 3.3.

$$T_{ideal} = \sum_{i=0}^{n-1} \tau_i + \sum_{i=0}^{n-1} (D^H_{i(in)} + D^H_{i(out)})\theta \tag{3.3}$$



Notice: 1: moving all input data from the main memory to the local memory of Kernel 1
2: moving all output data of Kernel 1 from its local memory to the main memory
2*: moving part of output data of Kernel 1 from its local memory to the main memory
3: moving all input data from the main memory to the local memory of Kernel 2
3*: moving part of input data from the main memory to the local memory of Kernel 2
4: moving all output data of Kernel 2 from its local memory to the main memory

Figure 3.4: The sequential diagrams for the baseline (left) and ideal execution model (right)

Compared to the baseline model, the execution time is reduced by $\Delta_{ideal} = T_b - T_{ideal} = \sum_{i=0}^{n-1}(D^K_{i(in)} + D^K_{i(out)})\theta$.

Figure 3.4 shows the sequential diagrams for the two execution scenarios of a hardware accelerator system which has one host processor and two kernels. The left diagram follows the baseline execution model while the right one follows the ideal execution model. Assume that part of the output from Kernel 1 is directly consumed by Kernel 2 (without any modification by the host). As shown in the diagrams, because only part of data output from Kernel 1 is copied back to the main memory (2* compared to 2) and part of data input for Kernel 2 is transferred from the main memory (3* compared to 3), the total execution time of the ideal execution model is shorter than the execution time of the baseline model.

The ideal execution model approach is different from the data communication optimization approaches presented in Section 2.4 which collect data for a kernel only when it is invoked. To approach this model, we propose design strategies to define an efficient hybrid interconnect for accelerator kernels using communication profiling. The hybrid interconnect, then, helps the kernel deliver its output. The objective of the strategies is to optimize the communication time while keeping interconnect resource usage minimal.

3.2.3. PARALLELIZING KERNEL PROCESSING

Given the fact that hardware accelerator systems have been increasingly used to address parallelizable and data intensive application domains [Ling et al., 2009] such as image or video processing [Cong and Zou, 2009], datacenter services [Putnam et al., 2014], etc., the parallelizing kernel processing can be used to further improve the system performance beside the proposed hybrid interconnect. Parallelism can be exploited at two different levels. Those are: data parallelism and instruction parallelism.

DATA PARALLELISM

Data parallelism is an execution scenario in which data is partitioned into segments, and concurrent processing kernels process those segments in parallel. In other words, one computationally intensive function can be accelerated by a number of concurrent kernels. Each kernel processes each data segment. Assume that a computationally intensive function has n accelerator kernels, data input for the function is partitioned into n segments. The reduction in processing time of this function compared to one accelerator kernel is $\Delta_{dp} = \frac{\tau_i(n-1)}{n} - O$ where τ_i is the time for processing the whole data with only one kernel and O is the overhead for data parallelism processing. This overhead depends on the application's algorithm and occurs because extra data needs to be processed to achieve the correct result for each segment [Gustafson, 1988]. Figure 3.5 shows a comparison between serial processing and data parallelism processing in which the function is accelerated by three different kernels (Kernel i_1, Kernel i_2, and Kernel_3) and data is partitioned into three segments.

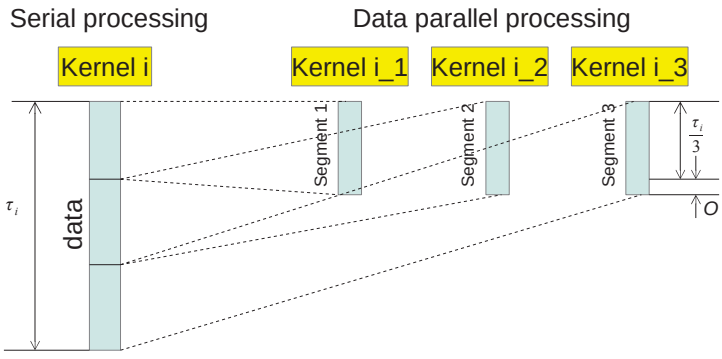


Figure 3.5: An example of data parallelism processing compared to serial processing

INSTRUCTION PARALLELISM

Instruction parallelism is an execution scenario in which the accelerator kernels of the functions form a pipeline to process a stream of data segments. Each kernel of the accelerated function establishes a pipeline stage. Data segments are streamed through those stages. Different from the serial execution scenario, in instruction parallelism, the kernels are working at the same time. The pipeline depth is the number of data segments processed. Assume that there are n processing stages, the total execution time in the serial scenario when the proposed hybrid interconnect takes care of the data communication between the kernels is evaluated by Equation 3.4.

$$T_{serial} = \sum_{i=0}^{n-1} \tau_i + \sum_{i=0}^{n-1} (D^H_{i(in)} + D^H_{i(out)})\theta \tag{3.4}$$

where $D^H_{i(in)}$ and $D^H_{i(out)}$ are the total amount of input data produced by the host and of output data consumed by the host.

When the pipeline scenario is exploited with depth m (assume that $m \geq n$), the total execution time is approximated by Equation 3.5

$$\begin{aligned} T_{pipeline} =& (\frac{\tau_0}{m} + O_0) + \max_{0\leq i\leq 1}(\frac{\tau_i}{m} + O_i) + ... + \max_{0\leq i\leq n-1}(\frac{\tau_i}{m} + O_i) \times (m-n+1) \\ &+ \max_{1\leq i\leq n-1}(\frac{\tau_i}{m} + O_i) + ... + \max_{n-2\leq i\leq n-1}(\frac{\tau_i}{m} + O_i) + (\frac{\tau_{n-1}}{m} + O_{n-1}) \\ &+ \sum_{i=0}^{n-1}(D^H_{i(in)} + D^H_{i(out)})\theta \end{aligned} \tag{3.5}$$

where O_i is the overhead explained in the previous section. The instruction parallelism is beneficial when $T_{pipeline} < T_{serial}$.

Figure 3.6 shows a comparison between a serial processing and a instruction parallelism processing. In the instruction parallelism processing, the number of pipeline stage, the number of kernels, is 3 ($n = 3$) while the pipeline depth, the number of data segments, is 4 ($m = 4$).

3.3. SUMMARY

In this chapter, we presented an overview of our approach to design a hybrid interconnect for a specific application using data communication profiling. The data communication driven quantitative execution model which is the goal of interconnect design is discussed. To further improve system performance, parallel

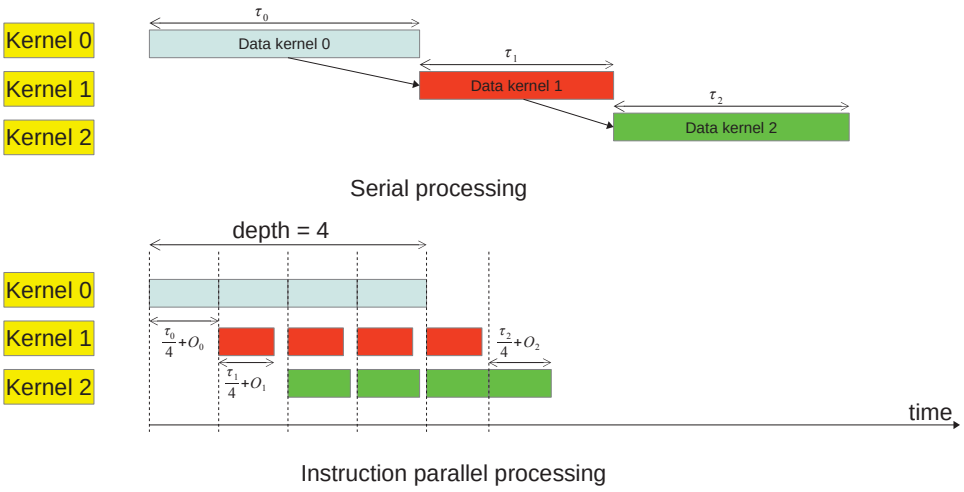


Figure 3.6: An example of instruction parallelism processing compared to serial processing

processing with support from the hybrid interconnect also presented. Compared to the baseline execution model, we aim to hide all the data communication among the accelerator kernels by delivering data from sources to destinations in parallel with kernels execution.

Note. The content of this chapter is partially based on the following papers:

1. *C. Pham-Quoc, Z. Al-Ars, K.L.M. Bertels*, **Automated Hybrid Interconnect Design for FPGA Accelerators Using Data Communication Profiling**, (May 2014), 28th International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2014), 19-23 May 2014, Phoenix, USA.

2. *C. Pham-Quoc, I. Ashraf, Z. Al-Ars, K.L.M. Bertels*, **Data Communication Driven Hybrid Interconnect Design for Heterogeneous Hardware Accelerator Systems**, (submitted), ACM Transactions on Reconfigurable Technology and Systems.

4

BUS-BASED INTERCONNECT WITH EXTENSIONS

In this chapter, we analyze an overview of different alternative interconnect solutions to improve system performance of a bus-based hardware accelerator system. A number of solutions are presented: direct memory access (DMA), crossbar, network-on-chip (NoC), as well as combinations of these. This chapter also proposes the analytical models to predict the performance for these solutions and implements them in practice. We profile the application to extract the data input for the analytical models.

4.1. INTRODUCTION

Although bus systems are usually used as the main communication infrastructure in many heterogeneous hardware accelerator systems due to their certain advantages [Guerrier and Greiner, 2000], they become inefficient when the number of cores rises. Moreover, in data intensive applications, such as multimedia computing, HD digital TVs, etc., a large amount of data needs to be transferred from core to core. Therefore, data communication is usually a primary anticipated bottleneck for system performance. Optimization of the interconnect taking the data communication into account is an essential demand.

In this chapter, we present an overview on the interconnect solutions used for hardware accelerator systems. To improve the performance of bus-based interconnects, a DMA, a crossbar, and a combination of both are used to consoli-

date the bus-based architecture. Moreover, NoC, a state-of-the-art interconnect approach, can be used to improve data communication behavior of hardware accelerators. In this work, we present the interconnect solution models to estimate the performance improvement of each interconnect compared to the bus-based interconnect. The experimental results show that the best system in terms of execution time and energy consumption is the system with a bus and a NoC, where the bus is used for the data exchange between the host and the hardware accelerators while the NoC is responsible for data communication among the hardware accelerators. Such system takes a toll of up to 20.7% additional hardware resource compared to the bus-based interconnect system.

The rest of the chapter is organized as follows. Section 4.2 briefly describes the related work. Section 4.3 presents in detail different interconnect solutions used in the heterogeneous hardware accelerator systems and their comparison. We implement experiments to validate the comparison between the interconnect solutions in Section 4.4. The discussion on the different interconnect solution is presented in Section 4.5. Finally, Section 4.6 summarizes the chapter.

4.2. RELATED WORK

In this section, we discuss different standard interconnect techniques as well as hardware accelerator systems that use a bus as the main communication infrastructure in the literature.

4.2.1. INTERCONNECT TECHNIQUES

Point-to-point interconnect is considered as the simplest interconnect solution for a system-on-chip (SoC). In a point-to-point interconnect architecture, the producer processing element (PE) is directly connected to the consumer PE. However, the biggest drawback of this architecture is the large number of wires required. This leads to difficulty in routing. Designs using this architecture are reported in [Dick, 1996], [ARM Limited, 2001].

The bus architecture is a low cost interconnect for SoCs. The two standard and well-known bus architectures are AMBA developed by ARM [ARM Limited, 1999] and CoreConnect developed by IBM [IBM, 1999]. Only CoreConnect has been adopted in Xilinx Virtex FPGA families. The main disadvantage of the bus architecture is the competition among modules (host processor, IO, memory controllers, etc) to access the bus introducing arbitrary latencies. This competition potentially degrades the performance of the system.

The crossbar is a well-known architecture for providing a high-performance and minimum latency interconnect. The main drawback of a crossbar is its cost. An $n \times n$ crossbar can quickly become prohibitively expensive as its cost increases by n^2. To reduce the cost, many studies focusing on application-specific crossbars have been reported such as in [Hur et al., 2007], [Murali et al., 2007].

In recent years, many Network-on-Chip architectures for FPGA have been reported such as DyNoC [Bobda et al., 2005], FLUX [Vassiliadis and Sourdis, 2006] and CuNoC [Jovanovic et al., 2007]. For low-latency applications-specific NoCs, driven by application task graphs, ReNoC [Stensgaard and Sparso, 2008] and Skiplinks [Jackson and Hollis, 2010] are used. Scalability is the main advantage of NoC. Moreover, NoCs are emerging as a high level interconnect solution ensuring parallelism and high performance. However, there are still several issues that need to be addressed such as power consumption and especially high area cost.

4.2.2. BUS-BASED HARDWARE ACCELERATOR SYSTEMS

Section 2.3 listed some bus-based hardware accelerator systems in academia and in commercial. Here we present in details some well-known hardware accelerator systems using a bus as the main communication infrastructure.

The *Molen* architecture [Vassiliadis et al., 2004] is a heterogeneous multicore system for software/hardware co-design. The Molen architecture consists of two types of processing elements (PEs): one *General Purpose Processor* (GPP) and one or more *Reconfigurable Processor(s)*, also so-called Custom Computing Unit(s) (CCUs). GPP has the main memory to contain application data while each CCU has each local memory (CCUMem) to contain its local data. The CCU exchanges parameters with GPP by exchange registers (CCUXreg) through an on-chip standard bus. While the GPP can access the main memory and the accelerator local memories, the accelerators can access only its local memory. The GPP and the accelerator local memories are also connected through an on-chip bus. When accelerator functions are needed, the GPP transfers data from the main memory to the local memory of the accelerator and copies the result back to the main memory when the accelerator finished.

A *Warp processor* [Lysecky and Vahid, 2009] consists of a main general purpose processor, an *efficient on chip profiler, an on-chip CAD module* (OCM) and *a warp-oriented FPGA* (w-FPGA). The main processor executes the software part of an application while the critical software regions are synthesized and mapped onto the w-FPGA. The selection, synthesis and mapping the critical software ker-

nels are done automatically by the profiler and the CAD module. The w-FPGA and the processor share the main data cache by using a mutually exclusive execution model. The main process, CAD module and the w-FPGA are connected together through an on-chip standard bus to configure the w-FGPA as well as to provide a mechanism for communication and synchronization between the main processor and the w-FPGA.

LegUp [Canis et al., 2013] is an open source high-level synthesis tool for FPGA-based processor/accelerators systems. The target system contains a processor connecting with custom hardware accelerators through a standard on-chip bus interface. The current version is implemented on the Altera Cyclone II FPGA with an Altera Avalon Bus as the interface for processor and accelerators communication. In this version, a shared memory architecture is used for exchanging variables between the processor and the accelerators. The shared memory uses an on-FPGA data cache and off-chip memory. The authors indicate that limitations of the bus system need to be further investigated.

The authors in [Schumacher et al., 2012] proposes *IMORC*, an infrastructure and architecture template that helps raising the level of abstraction to simplify the FGPA-based accelerator design. In the *IMORC* architecture, the computing cores are connected together through a multi-bus on-chip network. Each core has a number of communication ports which can be master or slave ports. One master port can connect with a number of slave ports via a bus. Beside the ports, the core comprises an execution unit and local memory. The execution unit can access the local memory and send message to other cores through the master port. The host processor which has a host interface core containing some communication ports communicates with the cores in the same protocol.

The PowerEN chip [Brown et al., 2011] [Heil et al., 2014] consists of 16 general purpose processors, two memory controllers, and a collection of hardware accelerators including Host Ethernet Adapter, Multi-Pattern Matching, Compression/Decompression, Cryptographic Data Mover, and XML Processing modules. Those components are connected together via a fabric called PBus. The PBus supports multiple module-to-module links and implements a snooping protocol to improve bandwidth. The accelerators communicate together through a memory buffer allocated in memory modules which are accessible by all the components.

4.3. DIFFERENT INTERCONNECT SOLUTIONS

In this section, we introduce different interconnect solutions used in heterogeneous hardware accelerators and give a comparison between them in terms of the total execution time of the hardware accelerators. In this work, we mainly focus on the data communication between the hardware accelerators.

4.3.1. ASSUMPTIONS AND DEFINITIONS

Hardware accelerator systems, such as Molen and target system in LegUp research, usually use a heterogeneous memory hierarchy in which the main memory is connected to the host while each hardware accelerator has its local memory to store data. In this work, we assume that the memory hierarchy is as follows:

- The host can access the main memory as well as the local memories of hardware accelerators through a standard on-chip bus; and
- The hardware accelerator kernel can access its local memory only.

In this chapter, we consider the hardware accelerator systems using a bus as the communication infrastructure and some consolidating interconnect techniques to improve system performance. We assume that a standard on-chip bus connects the local memories and the host together. We use the word "local memory" to refer to the local memory of a hardware accelerator. The word "main memory" is used for the main memory of the system which is connected to the host.

Before presenting different interconnects used in heterogeneous hardware accelerator systems, we need to define some equations used to compare the quality of the interconnect techniques. Beside the hardware accelerator kernel defined in Section 3.2.1, the following terminology is used:

- **Data communication** between two kernels is defined by $[HW_i \rightarrow HW_j : D_{ij}]$; where HW_i and HW_j are the producer and the consumer kernels, respectively, and D_{ij} is the total amount of data in bytes transferred from HW_i to HW_j.
- **The average time** taken by the host for transferring 1 byte from the main memory to a hardware accelerator local memory or vice versa via the bus is t_b, and the average time for transferring 1 byte from a hardware accelerator local memory to another one on the bus using DMA is t_d. These values are platform dependent, however $t_d < t_b$.

The "amount of data" mentioned in the data communication definition can be measured by using profiling tools such as the QUAD toolset [Ostadzadeh, 2012].

4.3.2. BUS-BASED INTERCONNECT

The bus system has some certain advantages compared with other interconnect techniques such as being compatible with most Intellectual Property (IP) blocks including host processors [Guerrier and Greiner, 2000]. Therefore, the bus system is considered as interconnect for many heterogeneous hardware accelerator systems. In these systems, the host uses the bus to transfer data between the main memory and the local memories. Figure 4.1 depicts an architecture using the bus system as interconnect.

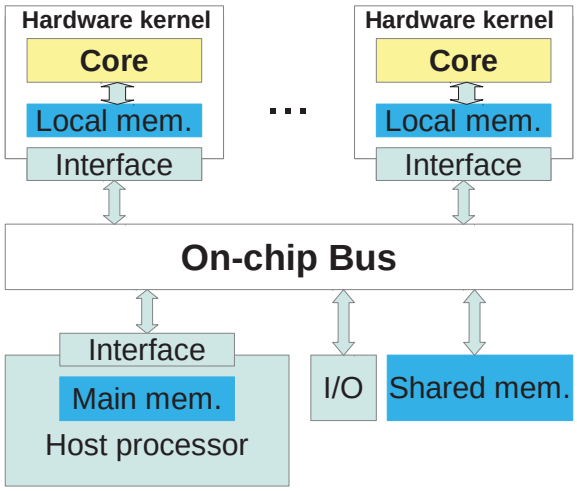


Figure 4.1: The bus is used as interconnect

Consider two accelerator kernels $HW_1(\tau_1, D^H_{1(in)}, D^K_{1(in)}, D^H_{1(out)}, D^K_{1(out)})$ and $HW_2(\tau_2, D^H_{2(in)}, D^K_{2(in)}, D^H_{2(out)}, D^K_{2(out)})$ communicating together with the communication $[HW_1 \rightarrow HW_2 : D_{12}]$. In many hardware accelerator systems, whenever the hardware accelerator is invoked, the host transfers input data from the main memory to the local memory. The kernel is executed right after all the required data is available in the local memory. Finally, the host copies the result of the hardware accelerator from the local memory to the main memory when the kernel is finished. Following these steps, the total execution time of the two hardware accelerators is shown in Equation 4.1. We refer to this model as the bus-

based model to which we compare other interconnect solutions.

$$T_b = \tau_1 + \tau_2 + (D_{1(in)} + D_{1(out)} + D_{2(in)} + D_{2(out)})\, t_b \tag{4.1}$$

where $D_{i(in)} = D^K_{i(in)} + D^H_{i(in)}$ and $D_{i(out)} = D^K_{i(out)} + D^H_{i(out)}$. We distinguish data from the host and from the kernels in this equation to compare to the other interconnect solutions presented later.

The main advantage of the bus-based interconnect is that the system is simple. The bus-based system can be implemented on most hardware platforms. However, the biggest disadvantage of this system is that the communication between hardware accelerators is not taken directly into consideration but has to go through the main memory. This leads to a high volume of data needed to be transferred through the bus. Additionally, the data movement performed by the host through the bus is usually very slow. The higher the amount of data communication is performed, the lower system performance is achieved.

In the next sections, we introduce techniques used to consolidate the bus to improve the performance of such systems.

4.3.3. BUS-BASED WITH A CONSOLIDATION OF A DMA

DMA is a technique that allows to access system memory independently of the host. DMA is usually shared the bus with the host and the local memories. The main advantage of DMA is that while DMA transfers data, the host can do other work. Moreover, DMA usually takes less time than the host for moving the same amount of data. The main disadvantage of DMA is the bus competition because it shares the bus with the host and the local memories. In addition, hardware resource overhead is also a disadvantage of DMA. Figure 4.2 depicts an architecture using the bus system with a consolidation of the DMA as interconnect.

In this solution, a DMA is used to consolidate the bus. DMA is responsible for transferring data from one local memory to another local memory. Different from the bus-based model, a communication profiling is used to improve the data communication operation. Consider the two above hardware accelerator kernels HW_1 and HW_2, the output $D^K_{1(out)}$ and $D^K_{2(out)}$ of the hardware accelerators are transferred to other hardware accelerators by the DMA rather than being written back to the main memory. In other words, the host is only responsible for transferring $D^H_{1(in)}$ and $D^H_{2(in)}$ from the main memory to the local memories as well as the result $D^H_{1(out)}$ and $D^H_{2(out)}$ from the local memories to the main memory. Other data movement is performed by the DMA. Following this way, the total

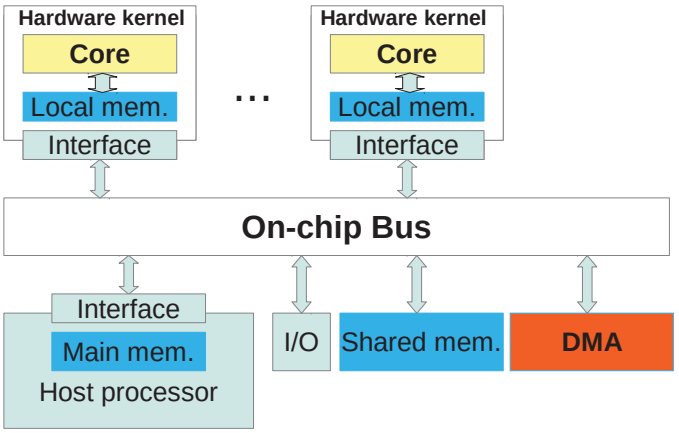


Figure 4.2: The DMA is used as a consolidation to the bus

execution time for the two hardware accelerators is as follows (Equation 4.2).

$$T_{dma} = \tau_1 + \tau_2 + (D^H_{1(in)} + D^H_{1(out)} + D^H_{2(in)} + D^H_{2(out)})\,t_b + (D^K_{1(in)} + D^K_{2(in)})\,t_d \quad (4.2)$$

The time needed to transfer the results of HW_1 and HW_2 (i.e., $D^K_{1(out)}$ and $D^K_{2(out)}$) to other local memories is not considered in this equation since it is taken into account by the execution time of the other hardware accelerators. In other words, they are transferred to the local memories of the consumed kernels by the DMA whenever the consumed kernels are invoked.

The total reduction in time compared to the bus-based model is as follows.

$$\Delta_{dma} = (D^K_{1(out)} + D^K_{2(out)})\,t_b + (D^K_{1(in)} + D^K_{2(in)})(t_b - t_d) \quad (4.3)$$

where $(D^K_{1(out)} + D^K_{2(out)})\,t_b$ is the reduction time because those outputs are not need to transferred back to the main memory while $(D^K_{1(in)} + D^K_{2(in)})(t_b - t_d)$ is the reduction in time because the DMA takes care those data inputs movement instead of the host.

4.3.4. BUS-BASED WITH A CONSOLIDATION OF A CROSSBAR

Crossbar is a high-performance and minimum latency interconnect technique. Although the cost of crossbar increases by n^2 where n is the number of inputs, small crossbar is area-efficient and delay-optimized. In this model, we consider a

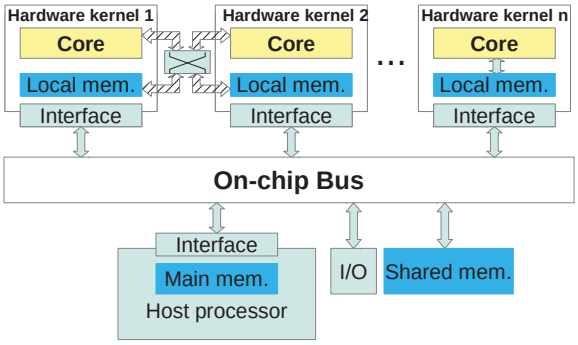


Figure 4.3: The crossbar is used as a consolidation to the bus

2×2 crossbar to share the local memories of the two hardware accelerators which communicate together. Figure 4.3 depicts the system using a crossbar as a consolidation to the bus. The main advantage of the crossbar is that there is no need to move data between the two hardware accelerators connected to the crossbar. The crossbar does not introduce any communication overhead because data is not needed to encode and decode when transferred through the crossbar. The main disadvantage of the crossbar is the additional hardware overhead for the crossbar logic.

Consider the 2 hardware accelerators HW_1 and HW_2 above. With the crossbar, HW_1 can access not only its local memory but also the local memory of HW_2. Therefore, neither the host nor the DMA is needed to transfer D_{12} from the local memory of HW_1 to the local memory of HW_2. The host is responsible for transferring other data. Based on this model, the total execution time of the two hardware accelerators is as follows.

$$T_{xbar} = \tau_1 + \tau_2 + (D_{1(in)} + D_{1(out)} + D_{2(in)} + D_{2(out)} - 2D_{12})\,t_b \tag{4.4}$$

where $D_{i(in)} = D^K_{i(in)} + D^H_{i(in)}$ and $D_{i(out)} = D^K_{i(out)} + D^H_{i(out)}$.

The total reduction in time compared to the bus-based model is as follows.

$$\Delta_{xbar} = 2D_{12}\,t_b \tag{4.5}$$

In this model, the reduction in time is $2D_{12}\,t_b$ because the host does not need to copy D_{12} from the local memory of HW_1 to the main memory as well as does not need to copy this data from the main memory to the local memory of HW_2 when it is invoked.

4.3.5. BUS-BASED WITH BOTH A DMA AND A CROSSBAR

Due to the advantages of both the DMA and the crossbar, they can be considered as consolidations to the bus to improve system performance at the same time. Consider a hardware accelerator system consisting of three hardware accelerators (as illustrated in Figure 4.4) in which two hardware accelerators HW_1 and HW_2 using the crossbar to share their local memory and the DMA is used for other data communication.

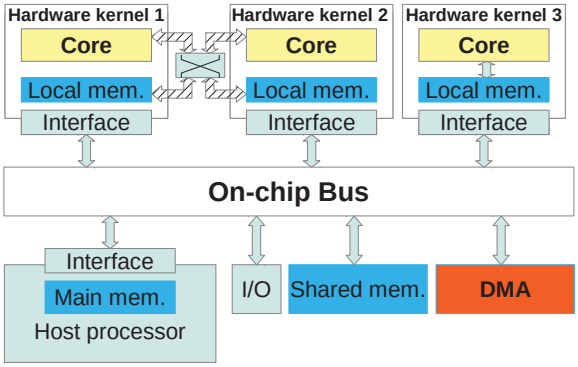


Figure 4.4: The DMA and the crossbar are used as consolidations to the bus

Assume that data communication between HW_1 and HW_2 is $[HW_1 \rightarrow HW_2 : D_{12}]$. With the DMA, the data input for the hardware accelerators from the other hardware accelerators (i.e., $D^K_{1(in)}$, $D^K_{2(in)}$, and $D^K_{3(in)}$), except D_{12}, are done by DMA. The host only takes care data produced and consumed by the host (i.e., $D^H_{i(in)}$ and $D^H_{i(out)}$). The host can do other work while the DMA performs the data movement. The total execution time of the three hardware accelerators is as follows (Equation 4.6).

$$T_{dma\&xbar} = \tau_1 + \tau_2 + \tau_3 + \sum_{i=1}^{3}(D^H_{i(in)} + D^H_{i(out)})\,t_b + (\sum_{i=1}^{3} D^K_{i(in)} - D_{12})\,t_d \qquad (4.6)$$

The total reduction in time compared to the bus-based model is as follows.

$$\Delta_{dma\&xbar} = \sum_{i=1}^{3} D^K_{i(out)}\,t_g + \sum_{i=1}^{3} D^K_{i(in)}(t_b - t_d) + D_{12}\,t_d \qquad (4.7)$$

In Equation 4.7, the first value $\sum_{i=1}^{3} D^K_{i(out)}\,t_g$ is reduced because the output of the hardware accelerators consumed by other hardware accelerators does not need

to be transferred back to the main memory. The second value $\sum_{i=1}^{3} D^{K}_{i(in)}(t_b - t_d)$ is the total time reduction when the DMA takes care the data movement from one local memory to another local memory instead of the host. The final value contribute to the equation because we do not need to move this data when the crossbar is used to share the local memories of HW_1 and HW_2.

4.3.6. NOC-BASED INTERCONNECT

NoC has been emerging as a high level interconnect solution ensuring parallelism and high performance. Although there are some certain disadvantages such as area overhead and latency [Guerrier and Greiner, 2000], a well designed NoC can be used as the interconnect among the hardware accelerators. In this model, we use both the bus and the NoC as the interconnect. The NoC is used to transfer data from one local memory to another while the bus is used to exchange data between the host and the hardware accelerators. Figure 4.5 shows a system using a NoC as interconnect of the hardware accelerators. Using only the NoC as interconnect is an alternative solution. However, this solution will incur a higher hardware overhead for the network interface at the host and higher delay in the communication between the host and the local memory compared to the bus.

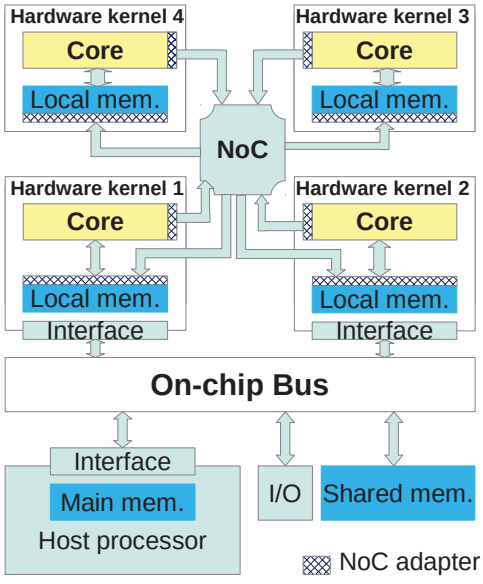


Figure 4.5: The NoC is used as interconnect of the hardware accelerators

With the NoC, the communication among the hardware accelerators is done in parallel with their execution. In other words, the output of one hardware accelerator is sent directly to the local memory of the consuming hardware accelerator through the NoC. Therefore, neither host nor DMA is required for data movement among the local memories. Consider a hardware accelerators with n hardware accelerators. The total execution time of the n hardware accelerators is as follows (Equation 4.8).

$$T_{noc} = \sum_{i=1}^{n} [\tau_i + (D^H_{i(in)} + D^H_{i(out)})\, t_b] \tag{4.8}$$

The total reduction in time compared to the baseline model is as follows.

$$\Delta_{noc} = \sum_{i=1}^{n} (D^K_{i(in)} + D^K_{i(out)})\, t_b \tag{4.9}$$

Since, data communication among the hardware accelerator kernels $[HW_i \rightarrow HW_j : D_{ij}]$ is done by the NoC in parallel with the kernels execution, the data communication is hidden. Therefore, the reduction in time Δ_{noc} is achieved in Equation 4.9.

However, the compatibility of the NoC and the hardware accelerators as well as the local memories needs to be addressed. The network interfaces should be developed to encapsulate the data and address generated by the hardware accelerators to the network packets at the hardware accelerator side and to decode the network packets to the data and address at the local memory side.

4.4. EXPERIMENTS

In this section, we present the experimental results using the aforementioned interconnect solutions. Based on those results, we compare the presented models and the execution time on a real hardware accelerator platform. Other system aspects such as hardware resource usage and energy consumption are also analyzed in this section.

4.4.1. EXPERIMENTAL SETUP

Before present the experimental results, we introduce the application and the way we implement the experiment considering all the aforementioned interconnect solutions. We use the Molen architecture as the base system. Xilinx ML510

board [Xilinx, 2009] containing a xc5vfx130t FPGA device is used as our hardware system. In this experiment, we use the jpeg application from the powerstone benchmark [Scott et al., 1998]. The QUAD toolset is used to generate data communication profiling for the application first. We then choose the most suitable functions to accelerate on hardware. Figure 4.6 shows the communication profiling graph for the jpeg application.

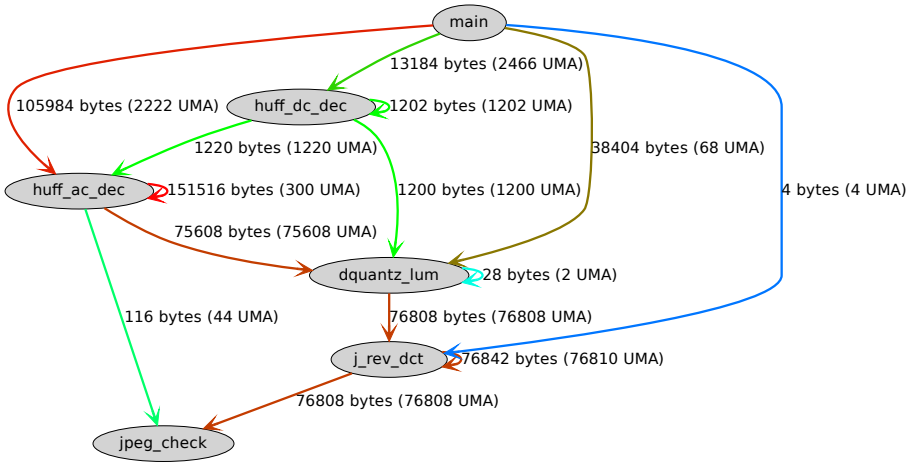


Figure 4.6: The communication profiling graph generated by QUAD tool for the jpeg application

In this experiment, 4 functions (*huff_ac_dec*, *huff_dc_dec*, *dquantz_lum* and *j_rev_dct*) are accelerated on hardware. The application is implemented using the Molen architecture first. The DWARV tool [Nane et al., 2012] is used to synthesize the functions from C code to VHDL code. In the Molen architecture, only the bus system is used as interconnect. The host is the PowerPC embedded in the FPGA device and hardware accelerators are mapped on to the reconfigurable area. The PowerPC is run at 400MHz while the hardware accelerators are executed at 100MHz. Block RAMs (BRAMs) are used as local memories. We then extend the system with the DMA, the crossbar, both the DMA and the crossbar, and the NoC.

In the extended systems, we develop our 2×2 crossbar to share the local memories of the two hardware accelerators as depicted in Figure 4.3. The Xilinx DMA IP core is used for DMA. A 3×2 NoC developed by Karlsruhe Institute of Technology, Germany [Heisswolf et al., 2012] is adapted as the NoC in the exper-

Table 4.1: Hardware resource utilization (#LUTs/#Registers) for each interconnect component and the frequency

Component	Resource utilization	Max. frequency
Bus	1048/188	345.8MHz
DMA	700/556	252.7MHz
Crossbar	201/200	N/A
NoC	1854/2122	150MHz
NI HW Accelerator	396/426	422.5MHz
NI local memory	60/114	874.2MHz

iment. We implement network interfaces (NIs) for the communication between the hardware accelerators as well as the local memories and the NoC. Table 4.1 presents the hardware resource utilization for each interconnect component and the maximum frequency.

4.4.2. EXPERIMENTAL RESULTS

In this section, we present the results for the jpeg application with different interconnect scenarios. We name the scenarios as Bus-based, DMA, Crossbar, DMA + Crossbar, and NoC-based for the bus-based interconnect, bus with a DMA, bus with a crossbar, bus with both DMA and crossbar, and NoC-based interconnect, respectively. In the jpeg application, we use two crossbars between *huff_ac_dec* and *huff_dc_dec* as well as between *dquantz_lum* and *j_rev_dct*.

Table 4.2 shows the computation time, the communication time, and the total execution time for the hardware accelerators of the jpeg application. These numbers are measured by the real execution using the FPGA board mentioned above. The computation time is the time for the hardware accelerator processing input data while the communication time is the time for data movement between components. The execution total time of a hardware accelerator is the sum of the computation time and communication time. As shown in the table, the computation time does not change in different scenarios. The NoC-based scenario is the most efficient interconnect since it reduces the communication time by 74.3% compared to the bus-based model. Hence, the total execution time in the NoC-based scenario results in a 2.4× speed-up compared to the bus-based scenario. Based on the models presented in Section 4.3 and the information from the communication profiling graph in Figure 4.6, the communication time of hardware accelerators for each scenarios is computed theoretically. This theoretical communication time is shown in Figure 4.7 normalized to the soft-

Table 4.2: Computation, communication and total execution time of hardware accelerators

Scenario	Computation	Communication	Total
Bus-based	2.07ms	7.52ms	9.59ms
DMA	2.07ms	2.54ms	4.61ms
Crossbar	2.07ms	2.87ms	4.94ms
DMA+Crossbar	2.07ms	2.20ms	4.27ms
NoC-based	2.07ms	1.93ms	4.00ms

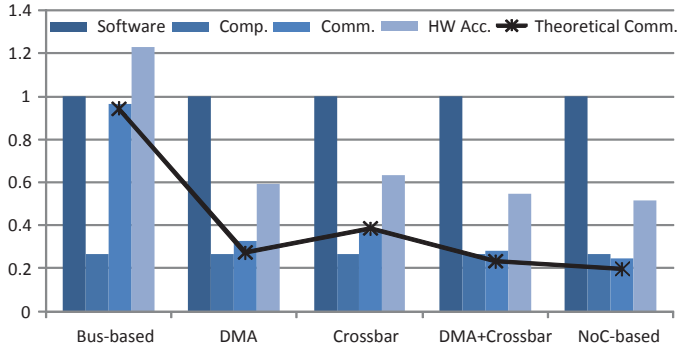


Figure 4.7: Comparison between computation (Comp.), communication (Comm.), hardware accelerator execution (HW Acc.), and theoretical communication (Theoretical Comm.) times normalized to software time

ware time of the hardware accelerators. The figure also compares the execution time normalized to the software time of the hardware accelerators in different scenarios. As shown in the figure, the theoretical communication time matches closely the measured communication time.

Table 4.3 gives the speed-up of the hardware accelerators and the overall application with respect to the software (the whole application is executed by the host only) and the bus-based model. The results show that the NoC-based model achieves a speed-up of up to 2.3× and 1.86× when compared to the bus-based model and the software, respectively. The table also shows that the performance of the bus-based model is even slower compared to the software due to the large communication time between the host and the hardware accelerators. Therefore, speed-ups compared to the bus-based model are larger than speed-ups compared to software. Figure 4.8 shows the speed-up of hardware accelerators in different scenarios with respect to the software and bus-based model.

Table 4.4 shows the hardware resource utilization for all scenarios. The results show that the NoC-based model requires additional 20.7% resources (which

Table 4.3: Speed-up of hardware accelerators and overall application compared to software and bus-based model

Scenarios	HW accelerators		Overall Application	
	w.r.t Software	w.r.t Bus-based	w.r.t Software	w.r.t bus-based
Bus-based	0.81×	1.00×	0.81×	1.00×
DMA	1.69×	2.08×	1.64×	2.02×
Crossbar	1.58×	1.94×	1.54×	1.90×
DMA+Crossbar	1.82×	2.25×	1.75×	2.16×
NoC-based	1.95×	2.40×	1.86×	2.30×

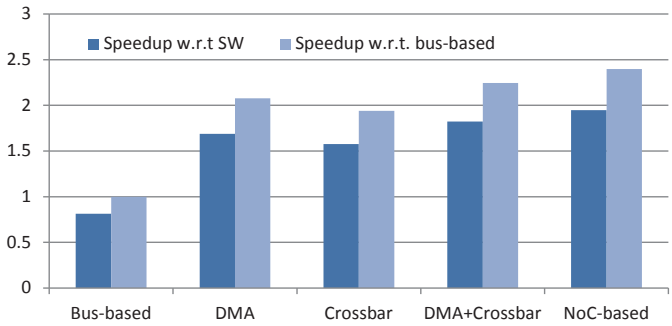


Figure 4.8: Speed-up of hardware accelerators with respect to software and bus-based model

Table 4.4: Hardware resource utilization (#LUTs/#Registers)

Scenario	Accelerator	Interconnect	Total
Bus-based	10707/11722	1048/188	11755/11910
DMA	10707/11722	1748/744	12455/12466
Crossbar	10707/11722	1249/388	11956/12110
DMA+Crossbar	10707/11722	1949/944	12656/12666
NoC-based	10707/11722	3490/2850	14197/14572

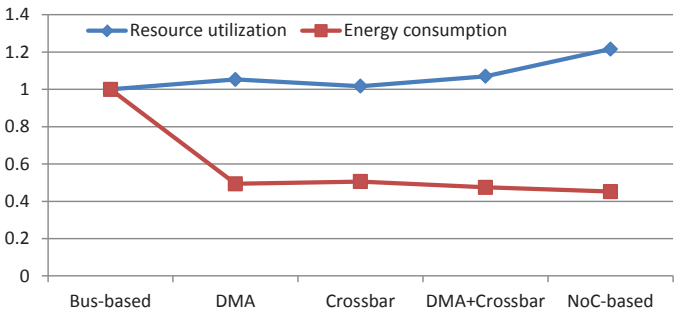


Figure 4.9: Comparison of resource utilization and energy consumption normalized to bus-based model

takes 2.9% of FPGA resources) compared to bus-based model. Figure 4.9 compares the hardware resource utilization and the energy consumption in different scenarios normalized to bus-based model. As shown in the figure, although the hardware resource utilization is the largest in NoC-based scenario, it is the smallest energy consumption scenario. The energy consumption is calculated as power consumption (estimated with Xilinx Power Analyzer) multiplied by the overall application execution time. For all scenarios, the power consumption is almost identical, with a slight increase following the increasing hardware resource utilization.

4.5. DISCUSSION

In Section 4.3, we presented the five different interconnect architectures for heterogeneous hardware accelerator systems. We implemented an experiment with the jpeg application. In this section, we discuss aspects of the interconnect architectures such as hardware resource utilization, hardware accelerator speed-up and the energy consumption.

Based on the models as well as the experiments, the NoC-based model is the best in terms of execution time but it uses the most hardware resource when compared to others. The more resources are used the more power consumption is needed. On the other hand, the bus-based with consolidation of DMA, crossbar, or a combination of both has a moderate improvement in speed-up and uses a limited amount of hardware resources.

Although modern devices, such as FPGA, contain a abundant amounts of resources, we have to choose trade off the number of resources and the price of the device. Moreover, the energy consumption is one of the main issues needed to be taken into consideration especially in battery-based systems. Energy consumption depends not only on power consumption but also the total execution time.

Based on the models, the designers can choose which interconnect solution is the most optimized for their systems. The designers have to choose trade off between the performance and the resource utilization. Depending on the requirements of the application as well as the resources available, the decision is made.

4.6. SUMMARY

This chapter presented an overview of interconnect solutions for hardware accelerator systems. The chapter investigated the impact of augmenting the solutions to an existing bus-based infrastructure. Performance models for bus-based, DMA, crossbar, DMA+crossbar, and NoC systems were discussed. The *jpeg* decoder is used for our case study in different scenarios using the presented interconnect solutions. Measurements made using these systems match the predicted analytical performance models. The NoC solution provides the highest performance achieving a speed-up of 2.4× compared to the bus-based system, and consuming the least amount of energy. At the same time, the NoC has the highest resource usage of up to 20.7% overhead.

Note. The content of this chapter is partially based on the following papers:

1. *C. Pham-Quoc, Z. Al-Ars, K.L.M. Bertels*, **Heterogeneous Hardware Accelerators Interconnect: An Overview**, NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2013), 25-27 June 2013, Torino, Italy

2. *C. Pham-Quoc, Z. Al-Ars, K.L.M. Bertels*, **Heterogeneous Hardware Accelerators Interconnect: An Overview**, 7th HiPEAC Workshop on Reconfigurable Computing (WRC 2013), 21 January 2013, Berlin, Germany

5

HEURISTIC COMMUNICATION-AWARE HARDWARE OPTIMIZATION

MULTICORE processing, especially heterogeneous multicore, is being increasingly used for data intensive processing in embedded systems. An important challenge in multicore processing is to efficiently get the data to the computing core that needs it. In order to have an efficient interconnect design for multicore architectures, a detailed profiling of data communication patterns is necessary. In this chapter, we present a heuristic-based approach to design an application-specific hardware accelerator system with a custom interconnect using quantitative data communication profiling information. A number of solutions are considered in this chapter. Those are crossbar-based shared local memory, DMA support parallel processing, local buffers, and hardware duplication. The ultimate goal is to have the most optimized custom interconnect design taking runtime communication pattern into account.

5.1. INTRODUCTION

As single-core microprocessors have reached the end of their scaling capabilities, hardware accelerator systems are becoming good platforms to process data intensive applications such as bio-informatics computing, multimedia computing, HD digital TVs, etc. The communication and the synchronization between

computing cores (microprocessor and hardware accelerators) are normally done through an interconnect network such as buses, Networks on Chip (NoC), etc. In data intensive applications, a large amount of data needs to be transferred from core to core. Therefore, data communication is usually a primary anticipated bottleneck for system performance [Altera, 2008; Becker et al., 2007; Donchev et al., 2006; Kavadias et al., 2010]. One important method to improve the performance of such systems is reducing data communication overhead.

Reducing data communication overhead can be done by increasing communication throughput or decreasing the amount of data movement from one memory to another. Examples of the former are [Altera, 2008; Becker et al., 2007; Donchev et al., 2006; Jackson and Hollis, 2010; Stensgaard and Sparso, 2008] and of the latter are [Kavadias et al., 2010; Papaefstathiou et al., 2007]. However, all the aforementioned works are based on static information of the application such as task graphs. The actual amount of data transferred between cores (which is responsible for data communication overhead) is not taken into consideration.

This highlights an important challenge in multicore processing, namely to efficiently get the data to the computing core that needs it. The goal is, of course, to hide the communication delay such that a performance improvement can still be observed. The resource allocation decision requires detailed and accurate information on the amount of data that is needed as input and what will be produced as output. Evidently, there are dependencies between computations since data produced by one core will be needed by another. To have an efficient allocation scheme where the communication delays can be hidden as much as possible, a detailed profile on the data communication patterns is necessary for which the most appropriate interconnect infrastructure can be generated. Such communication patterns can be specific for each application and could, therefore, lead to different types of interconnect. The work presented in this chapter is a first step towards a custom designed interconnect for an application. The ultimate goal is to change at runtime the interconnect infrastructure.

The main contributions of this chapter can be summarized as follows: 1. the introduction of a heuristic-based and detailed profile-driven interconnect design with an emphasis on runtime management; 2. the presentation of experimental results with seven different applications on a real FPGA platform; and 3. identification of the most suitable interconnect for each application domain in our experiment.

The rest of the chapter is organized as follows. Section 5.2 presents in de-

tail our approach to reduce data communication overhead and our proposed heuristic-based algorithm for a specific application using profiling information. Section 5.3 introduces our experiments with 7 different applications. Finally, Section 5.4 summarizes the chapter.

5.2. CUSTOM INTERCONNECT AND SYSTEM DESIGN

In this section, we introduce a heuristic-based algorithm to design an optimized application-specific custom interconnect. The heuristic-based algorithm uses data communication profiling information as a parameter to choose the most optimized interconnect solution.

5.2.1. OVERVIEW

As presented in Section 3.1.2, a detailed profiling information is required to define an efficient interconnect of accelerator kernels of a specific application. In this chapter, the proposed heuristic approach uses data communication profiling information of the application to create the interconnect for the accelerator kernels. The interconnect includes the solutions presented in the next section. The main purpose of the heuristic is to define an optimized interconnect in term of system performance while keeping the hardware resource usage minimal.

In this chapter, a hardware accelerator kernel is defined as in Section 3.2.1. Since our work target a generic hardware accelerator (i.e., it can be applied for an existing platform), we assume that the original hardware accelerator system consists of a host processor and the hardware fabric to implement the accelerator kernels. There is a predefined communication infrastructure for the communication among the host, the kernels, I/O, shared memory and other system components. The host has a main memory to store application data while the kernels have local memories to store kernels local data. The work in this chapter defines accelerator kernels and a custom interconnect for the kernels while keeping the original communication infrastructure because it is a predefined system backbone. Moreover, in some hardware accelerator system, this communication infrastructure is not reconfigurable.

5.2.2. DIFFERENT SOLUTIONS

In this section, we present a number of solutions to improve system performance, especially focusing on data communication behavior among the kernels as well as between the host and the kernels in the hardware accelerator system.

Solution 1 *Crossbar-based shared local memory*

The first interconnect we will use is a crossbar-based shared local memory in the case where two accelerators need to exchanges data. In this work, we use a crossbar for only two accelerators which communicate together to reduce the area overhead because the crossbar area cost increases quadratically. Figure 5.1(a) illustrates a simple system with the two hardware accelerators HW_1 and HW_2 sharing their local memories using a crossbar. When implemented on a real platform, the crossbar depends on the target hardware accelerator architecture. Figure 5.1(b) depicts the detailed structure of the crossbar used for the Molen architecture on an FPGA device.

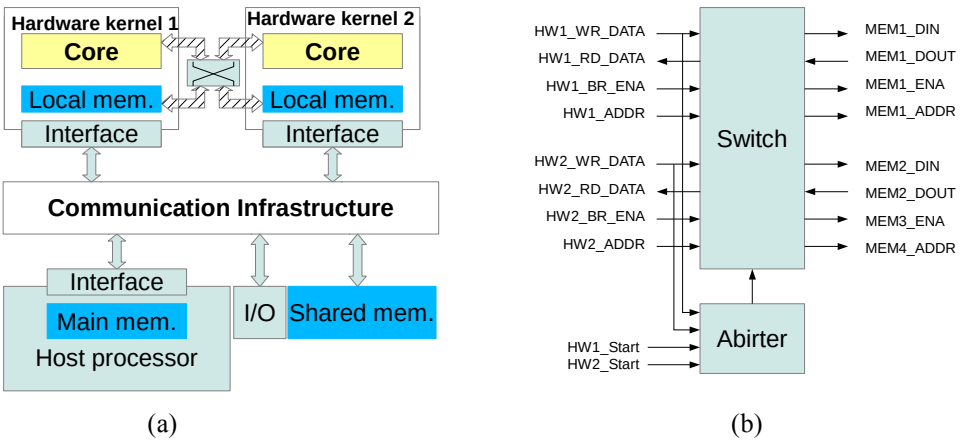


Figure 5.1: (a) HW_1 and HW_2 share their memories using a crossbar; (b) Structure of the crossbar for the Molen architecture

The QUAD tool identifies functions communicating together and how much data is transferred between them exactly. Based on this information, we can choose which functions should share their local memories via a crossbar. The execution time can be computed as follows.

Consider two hardware accelerators $HW_1(\tau_1, D^H_{1(in)}, D^K_{1(in)}, D^H_{1(out)}, D^K_{1(out)})$ and $HW_2(\tau_2, D^H_{2(in)}, D^K_{2(in)}, D^H_{2(out)}, D^K_{2(out)})$ which communicate together with data communication $[HW_1 \rightarrow HW_2 : D_{12}]$. According to the baseline system model presented in Section 3.2.1, the total execution time of the two hardware accelerators is as follows:

$$T_b = \tau_1 + \tau_2 + (D_{1(in)} + D_{1(out)} + D_{2(in)} + D_{2(out)})\theta \tag{5.1}$$

where $D_{i(in)} = D^K_{i(in)} + D^H_{i(in)}$ and $D_{i(out)} = D^K_{i(out)} + D^H_{i(out)}$; and θ is the average time to transfer one byte from the main memory to the local memory via the communication infrastructure.

Using the data communication profiling information, we know that D_{12} byte produced by HW_1 is consumed by HW_2. Therefore, it should be transferred directly from HW_1 to HW_2 instead of copied back to the main memory. One solution to reduce the execution time of the two hardware accelerators is to use direct memory access (DMA) to transfer data (D_{12} bytes) from the local memory of HW_1 to the local memory of HW_2. However, using DMA to transfer data from local memory not only has a hardware overhead but does not hide all data communication.

Through the crossbar, each hardware accelerator, HW_1 or HW_2, can access not only its own local memory but also the local memory of the another one. Therefore, HW_1 can write part of its result (D_{12} bytes) which is used by HW_2 to the local memory of HW_2. The host needs to transfer $D^H_{1(out)}$ bytes and ($D^K_{1(out)} - D_{12}$) bytes from the local memory of HW_1 to the main memory. The host can transfer this amount of data in parallel with the execution of HW_2 because this data is not used by HW_2. In other words, HW_2 can start sooner rather than waiting for the host copies $D^H_{1(out)}$ and ($D^K_{1(out)} - D_{12}$) back to the main memory. Similarly, the host can move $D^H_{2(in)}$ bytes from the main memory to the local memory of HW_2 in parallel with the execution of HW_1 if this data is available. However, this parallel behavior depends on the application and the communication infrastructure. In the following estimation equation, we do not take this parallel behavior into account.

Moreover, using the data communication profiling information, we can recognize data produced and consumed by the accelerator kernels only ($D^K_{i(in)}$ and $D^K_{i(out)}$). In other words, the data is not need to be copied back to the main memory because it is not used by the host. Therefore, the host can transfer the data directly from the producer to the consumer instead of copying back to the main memory. Consequently, the total execution time of the two hardware accelerators is as in Equation 5.2 which is asserted for almost all application.

$$T_{xbar} = \tau_1 + \tau_2 + (\sum_{i=1}^{2} D^H_{i(in)} + \sum_{i=1}^{2} D^H_{i(out)} + \sum_{i=1}^{2} D^K_{i(in)} - D_{12})\theta \tag{5.2}$$

In this equation, $\sum_{i=1}^{2} D^H_{i(in)}\theta$ and $\sum_{i=1}^{2} D^H_{i(out)}\theta$ are the time for transferring the accelerators input data produced by the host and for transferring the accelerators

output data consumed by the host, respectively. $(\sum_{i=1}^{2} D^{K}_{i(in)} - D_{12})\theta$ is the time for transferring input data for the accelerators produced by the other accelerators except D_{12} byte produced by HW_1 and consumed by HW_2 due to the shared local memory. In this equation, $D^{K}_{i(out)}$ does not contribute due to direct local memory to local memory data movement as explained above. With the crossbar, the total reduction time in comparison with the base system is $\Delta_{xbar} = \sum_{i=1}^{2} D^{K}_{i(out)}\theta + D_{12}\theta$.

Solution 2 *DMA support for parallel processing*

For some applications (multimedia) data can be processed as streaming input. Using this concept, we can reduce the data communication time by segmenting the input data and running the hardware accelerator on each data segment independently. When the data input is segmented, hardware accelerators execution and data movement can be executed in parallel. However, the host may be busy with other work or checking the execution of the kernels. One solution is to use DMA to transfer data directly from one local memory of a given hardware accelerator to another one via the communication infrastructure. Moreover, the average time for transfer one byte data from local memory to local memory by the DMA is usually lower than by the host.

Consider again the two hardware accelerators and the same communication that can execute in parallel on different segments, S_1 and S_2, of the input data. Assume that they cannot share their local memories by the crossbar. The processing flow following the pseudo code in Algorithm 1 can be applied to parallelize the data transfer from the main memory the hardware accelerator local memory and the processing of the hardware accelerators.

With this processing model, the total execution time of the two hardware accelerators is as in Equation 5.3 with an assumption that DMA transfer time is lower than hardware accelerator execution time.

$$T_p = \frac{\tau_1}{2} + max(\frac{\tau_1}{2}, \frac{\tau_2}{2} + \frac{D_{12}}{2}t_d) + \frac{\tau_2}{2} + (\frac{D^{H}_{1(in)}}{2} + \frac{D^{H}_{2(out)}}{2})\theta + \frac{D_{12}}{2}t_d + O \quad (5.3)$$

where O is the overhead for processing streaming input and t_d is the average time for transferring 1 byte from local memory to local memory be the DMA. The total reduction time in comparison with the base system is $\Delta_p = T_b - T_p = min(\frac{H_1}{2}, \frac{H_2}{2} + \frac{D_{12}}{2}t_d) + (\frac{D_{1i}}{2} + \frac{D_{2o}}{2})t_g - \frac{D_{12}}{2}t_d - O$.

Algorithm 1 Pipelining data communication

1: The host copies S_1 from the main memory to HW_1 local memory;
2: HW_1 processes S_1 while the host copies S_2 from the main memory to HW_1 local memory in parallel;
3: HW_1 processes S_2 while DMA transfers result of S_1 from HW_1 to HW_2 local memory and HW_2 processes the first segment right after DMA is finished;
4: DMA transfers result of S_2 from HW_1 to HW_2 local memory;
5: HW_2 processes the second segment while the host copies final result of the first segment from HW_2 local memory to the main memory in parallel;
6: the host copies final result of the second segment from HW_2 local memory to the main memory;

Solution 3 *Local buffer*

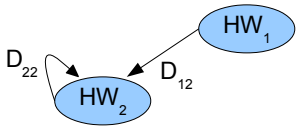


Figure 5.2: Local buffer at HW_2

Consider the two hardware accelerators HW_1 and HW_2 as in Solution 1. They communicate together with $[HW_1 \rightarrow HW_2 : D_{12}]$ as depicted in Figure 5.2. Assume that HW_1 is executed only one time while HW_2 is accelerated on hardware and iterated n ($n > 1$) times. HW_2 also communicates with itself with a communication $[HW_2 \rightarrow HW_2 : D_{22}]$. Due to the iteration of HW_2 using the same data, the part of data input for this hardware accelerator produced by HW_1 (D_{12} bytes) should be kept locally, which eliminates the need to transfer data from the main memory $n-1$ times (we need to transfer for the first time). Therefore, only $D^H_{2(in)}$ bytes need to be transferred from the main memory to local memory of HW_2 in each iteration. The total time of HW_2 is as follows.

$$T_{dl} = n\tau_2 + (D^H_{2(in)} + D^H_{2(out)})n\theta + D^K_{1(in)}\theta \tag{5.4}$$

The total reduction time in comparison with the based system is $\Delta_{dl} = D_{12}(n-1)\theta$.

Solution 4 *Hardware accelerator duplication*

In Solution 2, we introduced a way to parallelize the execution of the hardware accelerators. In case the data processed by a hardware accelerated function can be segmented and processed independently in parallel, we can replicate the hardware accelerator to further reduce the execution time. Assume that the hardware accelerator is duplicated twice, the data input of this accelerator is divided into two segments and each core processes each segment in parallel. A DMA is used to transfer the result of these hardware accelerators to others.

Consider a hardware accelerator $HW_1(\tau_1, D^H_{1(in)}, D^K_{1(in)}, D^H_{1(out)}, D^K_{1(out)})$, the execution time of the hardware accelerator in the non-duplication case and in the duplication case with DMA are as in Equation 5.5 and Equation 5.6, respectively.

$$T_{normal} = \tau_1 + (D_{1(in)} + D_{1(out)})\theta \tag{5.5}$$

where $D_{1(in)} = D^H_{1(in)} + D^K_{1(in)}$ and $D_{1(out)} = D^H_{1(in)} + D^K_{1(out)}$

$$T_{dp} = \frac{\tau_1}{2} + (D^H_{1(in)} + D^H_{1(in)})\theta + D^K_{1(in)} t_d + O \tag{5.6}$$

where O is the overhead for parallel processing. The time needed to transfer the results of HW_1 to other local memories (i.e., $D^K_{1(out)}$) is not considered in this equation since it is taken into account by the execution time of the consuming hardware accelerators. In other words, they are transferred to the local memories of the consuming kernels by the DMA whenever the consuming kernels are invoked. The total reduction time when compared to the base system is $\Delta_{dp} = \frac{\tau_1}{2} + D^K_{1(in)}(\theta - t_d) + D^K_{1(out)}\theta - O$.

5.2.3. HEURISTIC-BASED ALGORITHM

In the previous sections, we introduced different solutions to design a custom interconnect as well as a system using the quantitative data communication profiling. This section proposes a heuristic based algorithm to select the best and most suitable solution for each application. The pseudo code for the proposed algorithm is shown in Algorithm 2.

In this algorithm, the most computationally intensive functions suitable to implement on hardware (i.e., the function can be synthesized to a dedicated hardware circuit and the hardware fabric of the implemented platform is available for the function) are selected to accelerate on hardware. Currently, we choose only five functions as candidates for accelerating because our platform used to

Algorithm 2 Data communication profiling-driven design

Input: Application source code
Output: The most optimized interconnect

1: $L_{hw} \leftarrow$ List of the most computationally intensive functions suitable to implement on hardware;
2: $G \leftarrow$ Quantitative data communication graph for functions in L_{hw};
3: $G \leftarrow$ Sort G in decreasing amount of data transfer order;
4: Calculate Δ_{dp} as described in Solution 4 for the most computationally intensive hardware accelerator;
5: **if** $\Delta_{dp} > 0$ **then**
6: Apply the solution in Solution 4
7: **end if**
8: **for** each function in L_{hw} **do**
9: Check for iteration (Figure 5.2) and apply the Local buffer solution;
10: **end for**
11: **for** each data communication in G **do**
12: **if** the producer and the consumer can be executed in parallel **then**
13: Apply the solution in Solution 2;
14: **else if** The crossbar solution is applied to the producer or the consumer **then**
15: Use DMA to transfer data from the producer to the consumer
16: **else**
17: Apply solution in Solution 1;
18: **end if**
19: **end for**
20: **return** A hardware accelerator system with the most optimized interconnect

do experiments can support up to five hardware accelerators. Only the most computationally intensive hardware accelerator is considered for the hardware duplication solution. The QUAD profiling tool is used to identify the communication among the hardware accelerated functions. Based on this information, the algorithm examines the local buffer characteristic of the functions first. Then the interconnect solutions presented above are considered for each data communication between two hardware accelerators.

5.3. EXPERIMENTS

In this section, we present our experiment on the Molen architecture using an FPGA board. We firstly introduce the hardware platform used to implement our

experiment. A case study with the Canny edge detection application using the proposed heuristic approach illustrates our work. Finally, our experimental results with 7 different applications are given.

5.3.1. EXPERIMENTAL SETUP

In this work, we use the Molen architecture as the experimental platform. The Molen system is implemented on the Xilinx ML510 [Xilinx, 2009] board which contains a xc5vfx130t-ff1738 FPGA device. The embedded hardwired PowerPC is used as the host and the hardware accelerators are mapped onto the reconfigurable area. The main memory is the off-chip SDRAM connected to the PowerPC through a high performance IP core from Xilinx. The local memories of hardware accelerators are the on-chip Block RAMs (BRAMs). The Xilinx PLB bus is used as the system communication infrastructure. The PowerPC and the hardware accelerators are running at 400MHz and 100MHz, respectively. The Xilinx XPS DMA IP core is used as the DMA. In our experiment, the Molen architecture can support up to five different hardware accelerators due to limited FPGA resource.

We use the *gprof* profiling tool [Graham et al., 1982] to identify which part of the application takes most of the execution time. The *gprof* profiling tool provides a function call graph as well as the percent of the execution time of each function. Based on this graph, we can recognize which functions should be accelerated on hardware in order to reduce the execution time of the whole application. Functions with high computational-intensity are good targets for acceleration. The *QUAD* toolset [Ostadzadeh, 2012], which provides a comprehensive overview of the data communication behavior of an application, is used to generate the amount of data transfer between the functions of the application.

5.3.2. CASE STUDY

The previous sections presented the hardware techniques and the design algorithm using data communication profiling provided by QUAD to obtain the most optimized hardware accelerator system with a custom interconnect for each application. In this section, we present a case study to clarify the introduced techniques as well as the design algorithm.

Our case study uses the Canny edge detection application. The Canny application [Canny, 1986] is a well-known edge detection algorithm. In this work, we use the implementation version provided by the University of South Florida [Florida, 1999]. A grayscale *PGM* image with resolution 100×133 pixels with 8 bits

per pixel is used in the experiment. The most time-consuming functions in the specific application are targeted for hardware accelerators and inputted to the DWARV tool [Nane et al., 2012] to generate VHDL descriptions. The next step is to use QUAD tool to generate a QDU (Quantitative Data Usage) graph. Figure 5.3 presents the QDU graph generated profiling tools for the Canny application.

The information from the *gprof* tool shows that functions *gaussian_smooth*, *derrivative_x_y*, *magnitude_x_y* and *non_max_supp* take the most percentage of the execution time. Therefore, they are targeted to accelerate on hardware. The name of hardware accelerator functions has added prefix "hw_" for distinguishing from software functions. The most computationally-intensive function *gaussian_smooth* takes around 5× longer than the second computationally-intensive function. The reduction in time of this function if it is duplicated is larger than 0, i.e., $\Delta_{dp} > 0$. Hence, Solution 4 is applied to this hardware accelerator. DMA is used to transfer output data from those accelerator kernels to other kernel (kernel of the *hw_derrivative_x_y* function). The communication among *hw_derrivative_x_y*, *hw_magnitude_x_y* and *non_max_supp* functions is investigated next. Two functions *derrivative_x_y* and *magnitude_x_y* can execute in parallel with different data segments. Hence, Solution 2. The DMA support for parallel processing solution is used to parallelize the execution of *hw_derrivative_x_y* and *hw_magnitude_x_y*. The crossbar is used to share the local memories of *hw_magnitude_x_y* and *hw_non_max_supp* accelerators because the amount of this data communication is larger than the amount of data communication between *hw_derrivative_x_y* and *hw_magnitude_x_y*. Finally, data communication between *hw_derrivative_x_y* and *hw_non_max_supp* is done by DMA.

The final version of the hardware accelerator system based on the Molen architecture using presented hardware techniques and proposed design rules for Canny application is shown in Figure 5.4.

Table 5.1 summarizes the hardware size and maximum frequency of our hardware accelerator kernels as well as our crossbar. Table 5.2 summarizes the total software time, the total hardware time and the speed-up of the accelerator kernels with respect to software. Software time of a function is execution time when the function is executed on the PowerPC processor at 400MHz while hardware time of a function is execution time of the corresponding accelerator kernel. Row 1 shows the total software time of the functions; Row 2 and Row 3 shows the hardware time of the accelerated functions without and with applying our design ap-

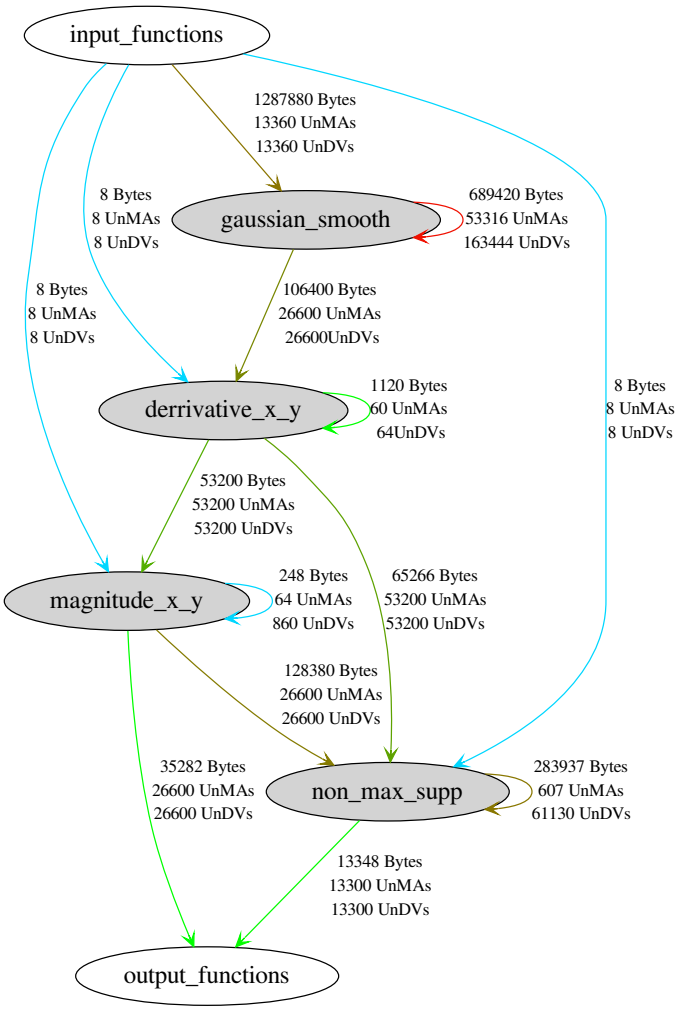


Figure 5.3: QUAD graph for the Canny edge detection application

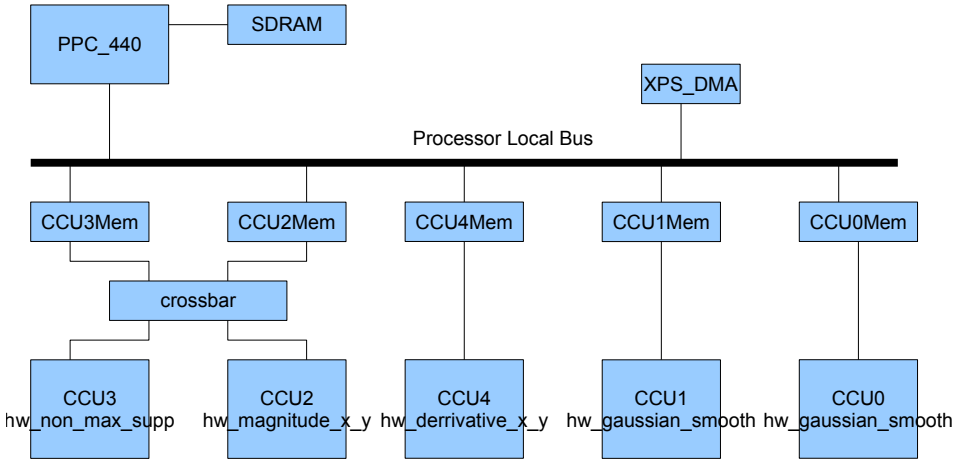


Figure 5.4: Final system for Canny based on the Molen architecture and proposed solutions

proach, respectively.

Table 5.1: Resource usage and maximum frequency of hardware modules

HW Module	**Resource (# of LUTs)**	**Max. frequency**
Crossbar	201	N/A
DMA	556	252.717MHz
hw_derrivative_x_y	1463	317.269MHz
hw_magnitude_x_y	971	388.342MHz
hw_gaussian_smooth	1938	264.460MHz
hw_non_max_supp	4959	313.908MHz

The maximum speed-up for hardware accelerator is up to 3.79× when the proposed design approach is applied. The overall application speed-up we can gain is 3.05×.

5.3.3. EXPERIMENTAL RESULTS

Beside the Canny edge detection application, we did experiments with 6 other different well-known applications. Those are the Susan edge detector [Smith, 1992] (with an implementation version of Oxford University), KLT feature tracker [Shi and Tomasi, 1994], Fluid simulation [Stam, 2003], the Blowfish application (a symmetric block cipher) from the CHStone benchmark [Hara et al., 2009], AES [SSL, 2012] and Bloom Filter [Christensen, 2012].

Table 5.2: Execution times of accelerated functions and speed-up compared to software

Scenario	**Execution time**	**Resource**	**Speed-up**
Software	16,723,007 (41.81ms)	N/A	N/A
Standard Molen	9,033,618 (22.58ms)	9331 LUTs	1.85×
Molen with rules	4,405,894 (11.02ms)	12026 LUTs	3.79×

Table 5.3: Interconnect techniques and hardware resource usage of applications

Application	HW techniques	# of LUTs	# of Kernels
Canny	Crossbar, Duplication, DMA	12026	5
SUSAN	Crossbar, DMA	21504	3
KLT	Crossbar, Duplication, DMA	6553	3
Fluid	Crossbar	12569	2
Blowfish	Crossbar, Local Buffer	16444	2
AES	Crossbar, Local Buffer	19544	2
Bloom filter	Crossbar, Local Buffer	1242	2

Table 5.3 present the detailed techniques used for the applications as well as the hardware resources and the number of kernels in each application. Column 2 in this table shows the custom interconnect techniques applied to each application. The crossbar technique is used for all applications in both the multimedia domain (the first four applications) and the cryptography domain (the last three applications). In the multimedia processing domain, the crossbar and the DMA techniques are frequently exploited. In the cryptography domain, only the crossbar and the local buffer techniques are used. Column 3 gives the total FPGA Look-Up Tables (LUTs) used by each application. Column 4 in this table shows the number of hardware accelerators used for each application. This column indicates that the multimedia processing applications have a tendency to use more hardware accelerator units, making them more suitable to accelerate on FPGAs compared to the cryptography applications.

Table 5.4 shows our experimental results for the seven different applications. Column 2 shows the overall application speed-up of Molen architecture with custom interconnect in comparison with software. Column 3 and column 4 show the speed-up of hardware accelerators (with respect to software) with and without the custom interconnect design.

As shown in Table 5.4, the speed-up of hardware accelerators and the overall application go up to 7.8× and 3.05×, respectively. Figure 5.5 shows the comparison of the speed-up of the Molen system with and without using our algorithm

Table 5.4: Application and kernel speed-ups with and without the custom interconnect w.r.t. software

Application	Application Speed-up	kernel speed-up w.r.t. SW	
		with custom interconnect	w/o custom interconnect
Canny	3.05×	3.79×	1.85×
SUSAN	2.51×	2.55×	2.02×
KLT	2.24×	7.8×	4.01×
Fluid simulation	1.50×	1.95×	1.45×
Blowfish	2.86×	3.02×	1.83×
AES	1.41×	2.81×	0.94×
Bloom filter	2.29×	3.05×	1.65×

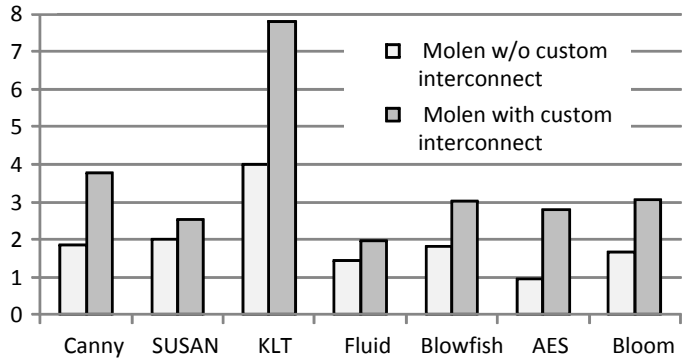


Figure 5.5: Speed-up (w.r.t software) of hardware accelerators using Molen platform with and without using custom interconnect

to choose the most optimized interconnect solutions. As shown in this figure, hardware accelerators which apply the communication profiling-driven acceleration solutions provide up to 2.98× execution time improvement in comparison with the accelerators that do not apply these acceleration solutions.

Figure 5.6 shows the contribution of each solution in the speed-up of each application when compared to the standard Molen architecture. From the figure, the crossbar-based shared local memory always contributes to the speed-up of applications. In the first four applications (Canny, Susan, KLT and Fluid), which belong to the multimedia processing domain, the different interconnects contribute in a different way to the overall speedup. The crossbar has a highest contribution in SUSAN, KLT and Fluid while the duplication gives the best per-

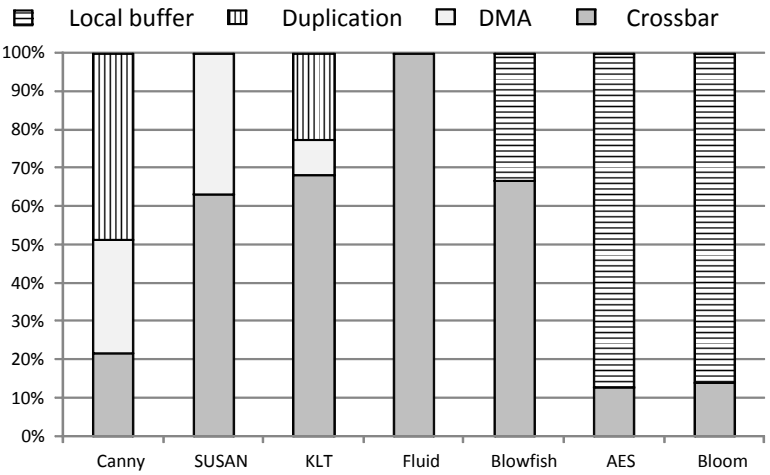


Figure 5.6: The contribution of each solution to the speed-up

formance in Canny. The DMA has a higher contribution to the speed-up when compared to the crossbar in Canny and when compared to the duplication in SUSAN. In the last three applications (which belong to the cryptography domain) the local buffer gives the best performance. The crossbar has a higher contribution when compared to the DMA and the duplication. Because the cryptography domain is very computation intensive on few data, it does not need a lot of communication. Therefore, a crossbar-based shared local memory accessible by all hardware accelerators will suffice.

5.4. SUMMARY

In this chapter, we presented a heuristic-based approach using a detailed data communication profiling to optimize an application-specific heterogeneous multicore system. The proposed approach mainly focuses on custom interconnect design. A heuristic-based algorithm is proposed to choose the most optimized interconnect solutions for each application. The algorithm uses data communication profiling information as a guidance parameter because this information allows the designer to make better founded decisions regarding the most appropriate interconnect. Our experimental results show that we can gain speed-up of hardware accelerators up to 7.8× in comparison with software and to 2.98× in comparison with the base system without using our approach. We also considered the contribution of each interconnect solution to each application as well as

to the application domain. Future research will investigate the possibility of tuning the interconnect at runtime. This runtime reconfigurability can be exploited evidently between applications but also within the execution of one application.

Note. The content of this chapter is partially based on the following papers:

1. *C. Pham-Quoc, Z. Al-Ars, K.L.M. Bertels*, **A Heuristic-based Communication-aware Hardware Optimization Approach in Heterogeneous Multicore Systems**, International Conference on ReConFigurable Computing and FPGAs (ReConFig 2012), 5-7 December 2012, Cancun, Mexico

2. *C. Pham-Quoc, Z. Al-Ars, K.L.M. Bertels*, **Rule-Based Data Communication Optimization Using Quantitative Communication Profiling**, International Conference on Field-Programmable Technology (FPT 2012), 10-12 December 2012, Seoul, Korea

6

AUTOMATED HYBRID INTERCONNECT DESIGN

THE communication infrastructure is an important component of a multicore system along with the computing cores and memories. A good interconnect design plays a key role in improving the performance of such systems. In this chapter, we introduce an automated interconnect design strategy to create an efficient custom interconnect for kernels in a hardware accelerator system. The main purpose of the hybrid interconnect is to accelerate the communication behavior of the kernels. Our custom interconnect includes a NoC, shared local memory solution, or both. Depending on the quantitative communication profiling of the application, the interconnect is built using our proposed custom interconnect design algorithm. An adaptive data communication-based mapping to connect the kernels to the interconnect is proposed to obtain a low hardware overhead and low latency interconnect. Compared to Chapter 5, this chapter considers a NoC instead of a DMA in the hybrid interconnect design.

6.1. INTRODUCTION

Evidently, each application has its own data communication patterns. The interconnect design for the application should take its communication patterns into account to define the most optimized interconnect for the application. In this chapter, we introduce our automated hybrid interconnect design. The main purpose is to improve the communication behavior of the kernels in an existing

accelerator system while keeping the amount of hardware resource usage for the interconnect as low as possible. In state-of-the-art approaches in the literature, input data required for kernel computation is fetched to its local memory (buffer) when the kernel is invoked, which delays the start-up of kernel calculations until the data is available. Although, there are some proposed solutions to tackle this delay, those solutions are ad-hoc solutions that are developed for specific architectures instead of a generic approach. Moreover, those approaches do not take the application data communication patterns into account.

In contrast to those approaches, our approach uses data communication profiling to create a custom interconnect for the kernels. The interconnect, then, helps deliver data from one kernel to the others as soon as possible, thereby hiding the data communication time needed for the kernel. Therefore, we approach the ideal execution model presented in Section 3.2.2. The ultimate goal is to have a tailored interconnect infrastructure which is dynamically configured. A custom interconnect design algorithm and an adaptive mapping function using a quantitative data communication profiling of the application are proposed to build the interconnect.

Our results in an embedded platform show that the proposed system achieves a speed-up of an overall application by up to 2.87× compared to a baseline system (a bus-based hardware accelerator system). We also managed to save up to 66.5% energy consumption. The experimental results on the Convey high performance computing platform show that a speed-up of an overall application by up to 1.55× compared to the baseline high performance computing system is obtained and up to 63% energy consumption is reduced.

The rest of this chapter is organized as follows. Section 6.2 presents in detail the mathematical modeling of system components and our proposed design strategy. In Section 6.3, experimental results show the benefit of the custom interconnect architecture using four experimental applications. Finally, Section 6.4 concludes the chapter.

6.2. AUTOMATED HYBRID INTERCONNECT DESIGN

To reach the ideal execution model, the data communication among the kernels needs to be hidden. In other words, the hybrid interconnect of the kernels transfers data from sources to destinations in parallel with the execution of the kernels. Therefore, in this section, our design approach driven by the data communication profiling is introduced in order to define the hybrid interconnect for

the kernels with optimized execution time and hardware resource usage. To derive a mathematical model for performance estimation, we denote communication between any two kernels of the system as $[HW_i \rightarrow HW_j : D_{ij}]$ in which HW_i sends D_{ij} byte to HW_j. This communication behavior can be extracted from the data communication profiling of the application.

6.2.1. MODELING SYSTEM COMPONENTS

In this section, we analyze the potential benefit of different interconnect choices as compared to the simple baseline system. We compute for each of the solutions what improvement it brings over the baseline model. We conclude by introducing a number of hybrid implementation and show the potential pay off based on the same execution model.

MODELING SHARED LOCAL MEMORY

We consider to share the local memories of two kernels in which one kernel (HW_i) only sends its $D^K_{i(out)}$ output to another kernel (HW_j) and HW_j only receives $D^K_{j(in)}$ input from HW_i (i.e., $[HW_i \rightarrow HW_j : D_{ij}]$ and $D^K_{i(out)} = D^K_{j(in)} = D_{ij}$). With the shared local memory, D_{ij} byte of data can be used without any movement. Hence, compared to the baseline model (presented in Section 3.2.1), communication time for this data segment is reduced by $\Delta_c = 2D_{ij}\theta$ (one time from the local memory of HW_i to the host and one time from the host to the local memory of HW_j).

When implemented on FPGA-based platforms, most accelerator systems use block RAM (BRAM) as the local memory. BRAMs in modern FPGA usually have only two ports while they may be accessed by three different components (the two communicating kernels and the host). Therefore, there are two practical solutions to implement the shared local memory. These solutions are chosen based on the communication pattern generated by the profiling tool.

- *Case 1*: if the receiving kernel HW_j communicates with the host (i.e., $D^H_{j(in)} \neq 0$ or $D^H_{j(out)} \neq 0$), one of the two BRAM ports is dedicated to communicate with the host (the same situation is reported in [Choi et al., 2012]). Therefore, a crossbar is used to share the local memories of the two kernels HW_i and HW_j. Kernel 1 and Kernel 2 in Figure 6.1 illustrate this case. The crossbar switches data from the cores to the corresponding local memory based on the address of data. The crossbar does not introduce any communication overhead because it does not change the structure of data. In other

words, we do not need to encode and decode data format.

- *Case 2*: if the receiving kernel HW_j does not communicate with the host (i.e., $D^H_{j(in)} = D^H_{j(out)} = 0$), HW_i and HW_j can share the local memories without the crossbar. Kernel 3 and Kernel 4 in Figure 6.1 illustrate this case.

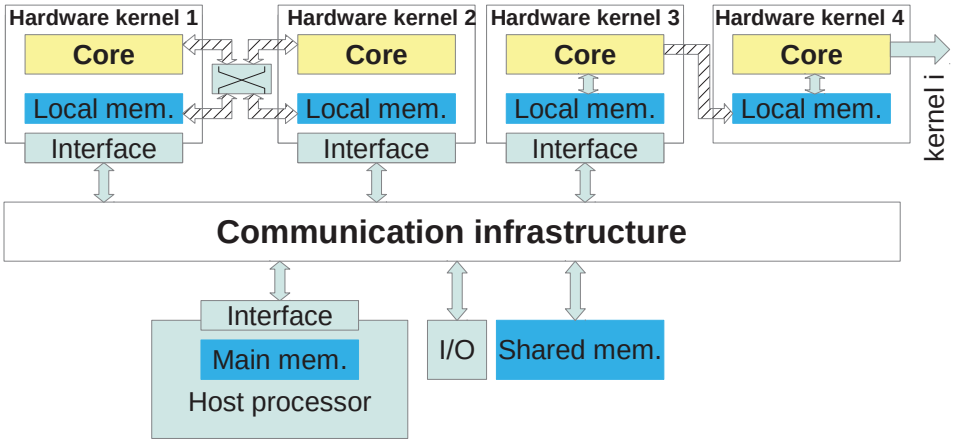


Figure 6.1: Shared local memory with and without crossbar in a hardware accelerator system.

MODELING NOC-BASED INTERCONNECT

NoCs are an established and widely used interconnect method providing parallelism and high performance. Although there are certain disadvantages such as area overhead and latency [Guerrier and Greiner, 2000], a well designed NoC can be used as the interconnect of the kernels. Figure 6.2 shows a hardware accelerator system in which the kernels use a NoC as their interconnect. An alternative solution is using only the NoC as interconnect of the whole system, i.e., the communication infrastructure in Figure 6.2 is eliminated and the host, I/O, the shared memory are directly connected to the NoC. However, this solution will incur a higher hardware overhead for the network adapters at the host and the I/O. Most hardware accelerator systems have a predefined communication infrastructure and predefined connections of the host, the shared memory, the I/O, etc., to the communication infrastructure. Adding a NoC to accelerate the communication behavior of the kernels is more suitable than modifying the whole system. Additionally, in some systems where the host is located on a separated chip from the kernels (such as the Convey architecture [Convey Computer,

2012]), the communication infrastructure of these systems is usually not reconfigurable.

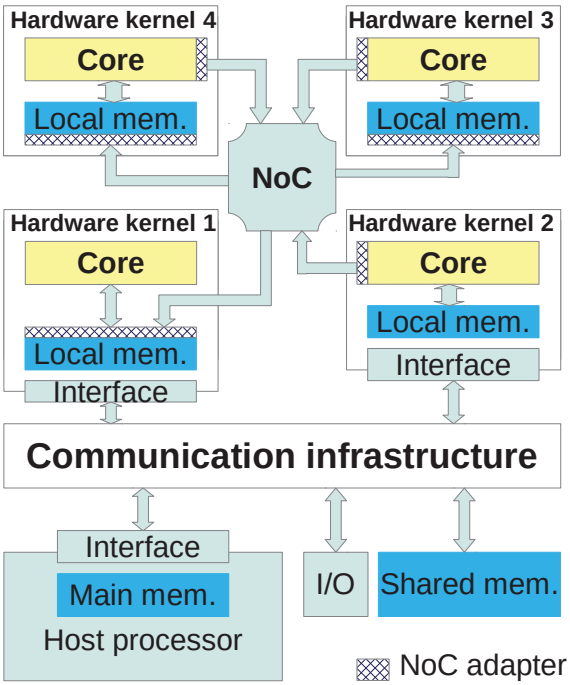


Figure 6.2: The NoC is used as interconnect of the kernels in a hardware accelerator system.

With the NoC, data communication among the kernels is done in parallel with their execution. In other words, data output of one kernel is sent directly to the local memories of the consuming kernels through the NoC as soon as it is available instead of storing in the local memory of the kernel. Hence, kernel HW_i does not need to collect $D^K_{i(in)}$ from the main memory and data output $D^K_{i(out)}$ is not copied back to the main memory when finished. Consequently, communication time of the kernels is hidden. Compared to the baseline model, the NoC reduces the execution time by $\Delta_n = \sum_{i=0}^{n-1}(D^K_{i(in)} + D^K_{i(out)})\theta$. Figure 6.3 illustrates the detailed execution of a hardware accelerator system consisting of three kernels using a NoC as the kernel interconnect. The three accelerator kernels in the illustrated system are K1 ($HW_1(\tau_1, D^H_{1(in)}, 0, D^H_{1(out)}, D^K_{1(out)})$), K2 ($HW_2(\tau_2, D^H_{2(in)}, D^K_{2(in)}, D^H_{2(out)}, D^K_{2(out)})$), and K3 ($HW_3(\tau_3, D^H_{3(in)}, D^K_{3(in)}, D^H_{3(out)}, 0)$). Phases 3, 4, 6, and 7 in the NoC-based interconnect system are shorter than in the baseline system due to data movement through the NoC. While all K1 out-

put ($D^H_{1(out)}$ and $D^K_{1(out)}$) is copied back to the main memory in the baseline system in Phase 3, only part of this output ($D^H_{1(out)}$) is copied to the main memory in the NoC-based interconnect system because data output consumed by K2 and K3 ($D^K_{1(out)}$) is transferred to K2 and K3 by the NoC in Phase 2 (parallel with K1 execution). Consequently, when K2 is invoked, only data input generated by functions in the host ($D^H_{2(in)}$) is moved from the main memory in Phase 4 in the NoC-based interconnect system rather than the whole data input ($D^H_{2(in)}$ and $D^K_{2(in)}$) as in the baseline system. Phase 6 and Phase 7 do the same behavior as already explained.

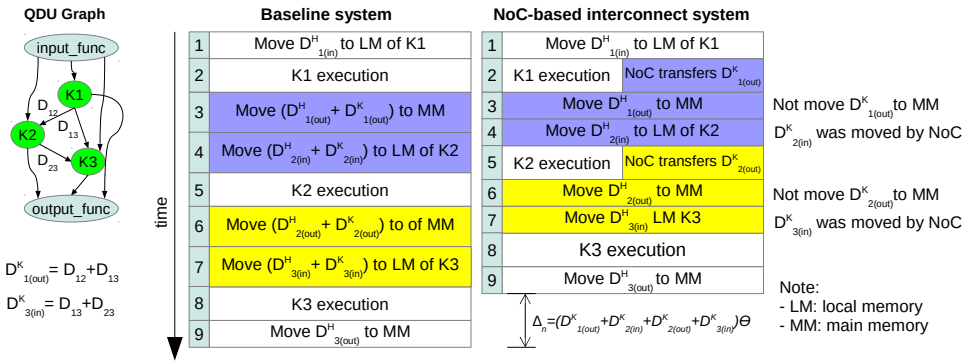


Figure 6.3: Illustrated NoC-based interconnect data communication for a hardware accelerator system.

In this NoC solution, the kernels and their local memories are connected to NoC routers through adapters. The NoC adapter encodes data input from the kernel into NoC packets and sends them to the NoC at the kernel side. At the local memory side, the NoC adapter decodes incoming packets and stores data to the local memory. The number of routers and adapters is a sum of the components connected to the NoC, i.e., the more components are connected to the NoC, the more routers and adapters are needed. The more routers and adapters are used, the larger resources are required. That is the reason why we consider a hybrid interconnect to have an optimized resource usage in this work rather than using only a NoC as the kernels' interconnect.

Assume that beside the shared local memory, there are still n kernels and their n local memories using a NoC as interconnect in a hardware accelerator system, it is not necessary to connect all the n kernels and n local memories to the NoC ($2n$ routers and adapters are required for the full connection). For example, in Figure 6.2, Kernel 1 and local memory in Kernel 2 are not connected

to the NoC because we assume that Kernel 1 does not send data output to any other kernel while Kernel 2 does not receive data input from any another kernel (i.e., $D^K_{1(out)} = 0$ and $D^K_{2(in)} = 0$). Therefore, based on the communication topology of each specific application, we define a different connection topology of the kernels and the local memories to the NoC so that the number of routers and adapters required is as low as possible. Details of this proposed adaptive mapping algorithm is presented in Section 6.2.3.

6.2.2. CUSTOM INTERCONNECT DESIGN

In this section, an automated design approach is proposed to define an efficient hybrid interconnect in terms of optimized communication time and low hardware resource usage. The hybrid interconnect consists of the shared local memory solution along with a NoC. Evidently, accelerator kernels and their communication behavior are different from one application to the other. Therefore, a specific application should have a specific hybrid interconnect to efficiently get data to the kernels that need it. To define the most optimized hybrid interconnect for the kernels, the interconnect solutions are chosen as the following ordering:

- The shared local memory (as presented in Section 6.2.1) is considered before the NoC solution because of the hardware resource usage. If the NoC is used instead of the shared local memory, we need four routers and four adapters (two for kernels and two for their local memories). Keeping in mind that the hardware resources usage for those routers and adapters is 5× larger than the hardware resources usage for the shared local memory solution (using a crossbar or directly sharing the local memory).
- All the remaining kernels and local memories use a NoC as interconnect.
- The proposed adaptive mapping function (presented later) will make a connection topology for the kernels and the local memories to the NoC and the communication infrastructure so that the number of required routers and adapters is minimal.

Algorithm 3 shows the pseudo code of the hybrid interconnect design algorithm. The result of the algorithm is a hybrid interconnect with the most optimized communication time while keeping the hardware resource usage for the interconnect as low as possible. The algorithm, first, selects functions which are the most computationally intensive and suitable for acceleration on the hardware fabric (i.e., those functions that can be implemented in hardware) (line

Algorithm 3 Custom interconnect design

Input: Application source code
Output: The most optimized interconnect

1: $L_{hw} \leftarrow$ List of the most computationally intensive functions suitable to implement on HW;
2: **for** each HW in L_{hw} **do**
3: **if** HW satisfies the data parallelism in Section 3.2.3 ($\Delta_{dp} > 0$) & resource is available **then**
4: Duplicate HW in L_{hw}
5: **end if**
6: **end for**
7: $G \leftarrow$ Quantitative data communication profiling for functions in L_{hw};
8: **for** each communication $[HW_i \rightarrow HW_j : D_{ij}]$ in G **do**
9: **if** $D^K_{i(out)} = D^K_{j(in)} = D_{ij}$ **then**
10: Apply the shared local memory solution for HW_i and HW_j
11: Remove HW_i from L_{hw}
12: **end if**
13: **end for**
14: Map all HW in L_{hw} to the NoC using adaptive mapping
15: Apply the instruction parallelism solution in Section 3.2.3 if possible

1). The most computationally intensive functions are considered for data parallelism solution if acceptable (i.e., if the most computationally intensive functions can be parallelized and the hardware resources are available, they will be replicated) (line 2-6). The algorithm, then, uses the QUAD toolset to generate a quantitative data communication profiling of the application (line 7). Based on this detailed profile, an efficient custom interconnect using the presented solutions is built. In line 8-14, the hybrid interconnect is built using the above mentioned ordering. The details of the mapping in line 14 will be discussed in Section 6.2.3. Finally, the instruction parallelism is considered to further reduce the execution time if acceptable (line 15).

6.2.3. ADAPTIVE MAPPING FUNCTION

As explained in Section 6.2.1, it is not necessary to map all the kernels and their local memories remaining after the shared local memory is considered to the NoC. Depending on the communication patterns, there are different ways to connect a kernel and its local memory to the NoC (to communicate with other

kernels) and the communication infrastructure (to communicate with the host). Therefore, the adaptive mapping function is introduced to define a connection of a kernel and its local memory to both the NoC and the communication infrastructure so that the number of required routers and adapters is minimal. The proposed adaptive mapping function used in Line 14 of Algorithm 3 is shown in Equation 6.1.

$$f : Communication \rightarrow Interconnect \tag{6.1}$$

where the $Communication$ and the $Interconnect$ values are defined below.

A kernel can receive data input from three different sources:

1. only from other kernels (R_1);
2. only from the host (R_2);
3. from both other kernels and the host (R_3).

Similarly, a kernel can send data output to three different destinations:

1. only to other kernels (S_1);
2. only to the host (S_2);
3. to both other kernels and the host (S_3).

Therefore, each kernel has nine different data communication topology cases as in Equation 6.2.

$$Communication = \{R_1, R_2, R_3\} \times \{S_1, S_2, S_3\} \tag{6.2}$$

There are two options for a connection between a kernel and the NoC

1. the kernel is not connected with the NoC (K_1);
2. the kernel is connected with the NoC (K_2).

Similarly, there are three options for a connection of a local memory with the communication infrastructure and the NoC

1. the local memory is connected to the communication infrastructure only (M_1);

2. the local memory is connected to the NoC only (M_2);

3. the local memory is connected to both (M_3).

Therefore, each kernel and its local memory have six different interconnect topology cases as in Equation 6.3.

$$Interconnect = \{K_1, K_2\} \times \{M_1, M_2, M_3\} \tag{6.3}$$

Table 6.1 shows the mapping of the communication topology to the interconnect topology. The interconnect value $\{K_1, M_2\}$ (the kernel is not connected to the NoC while its local memory is only connected to the NoC) is not feasible as the result of the accelerator kernel will be inaccessible by any other function.

Table 6.1: Adaptive mapping function

Communication	Interconnect
$\{R_1, S_1\}$	$\{K_2, M_2\}$
$\{R_1, S_2\}, \{R_3, S_2\}$	$\{K_1, M_3\}$
$\{R_1, S_3\}, \{R_3, S_1\}, \{R_3, S_3\}$	$\{K_2, M_3\}$
$\{R_2, S_1\}, \{R_2, S_3\}$	$\{K_2, M_1\}$
$\{R_2, S_2\}$	$\{K_1, M_1\}$

To reduce the NoC latency, a kernel and its communicating local memories should be mapped to the NoC routers in such a way that the distance of these routers is shortest. For instance, if a kernel is mapped to a router at the (x, y) coordinate then the ideal location for the local memory to which it communicates is either $(x-1, y)$, $(x+1, y)$, $(x, y-1)$, or $(x, y+1)$.

The objective of this mapping function is to define the most optimized topology in terms of HW resource usage. An alternative simpler solution is to map all the kernels and all their local memories to both the NoC and the system communication infrastructure. However, this mapping solution requires the maximum number of routers as well as network adapters. Different from other state-of-the-art mapping algorithms for an FPGA NoC-based system such as [Singh et al., 2010; Yu et al., 2010] which map application tasks to NoC only, our work considers to map both the kernels and the memory to both the NoC and the system communication infrastructure of the hardware accelerator system.

6.3. EXPERIMENTAL RESULTS

In this section, we present our experimental results. We validate our proposed interconnect design in both embedded systems and high performance computing systems. The Molen platform [Vassiliadis et al., 2004] is used as our embedded experimental platform while the Convey HC-2ex [Convey Computer, 2012] is used as the high performance computing system. In order to implement our proposed hybrid interconnect for the kernels, we develop a 2×2 crossbar for the shared local memory solution and adapt the NoC presented in [Heisswolf et al., 2012] into our systems.

The following sections present in detail our experimental result for each system.

6.3.1. EMBEDDED SYSTEM RESULTS

Even though, the Zynq board is a nice example of the Molen architecture, it was not available at the time of these experiments. Therefore, to build the Molen platform, a Xilinx ML510 [Xilinx, 2009] board containing an xc5vfx130t FPGA device is used. The embedded hardwired PowerPC processor acts as the host processor while accelerator kernels are mapped onto the reconfigurable area of the device. SDRAM memory connected directly to the PowerPC through a Xilinx core is the main memory of the system. While the host processor works at 400MHz, the kernels work at 100MHz. The Xilinx PLB bus is used as the communication infrastructure which connects the host processor, the kernels and other modules such as I/O, Interrupt, Timer, etc., together. The DWARV [Nane et al., 2012] compiler automatically generates the HDL description for the kernels from their C code. The system is synthesized with Xilinx ISE 13.2 without any manual optimization.

PERFORMANCE ANALYSIS

To validate the acceleration ability of the Molen platform, we first compare the execution time of the Molen system to software execution (using the PowerPC only). Please note that, almost all the embedded accelerator systems presented in Section 2.3 compare their systems against a host running at a low frequency, for example, 85MHz in [Lysecky and Vahid, 2009], 125MHz in [Ismail and Shannon, 2011], 100MHz in [Pilato et al., 2012], 75MHz in [Canis et al., 2013], etc., while the Molen system in this work is compared to a host running at 400MHz. Four applications are used for the following experiments. Those are the Canny edge detection application [Canny, 1986], the *jpeg* decoder application [Scott et al., 1998], KLT feature tracker [Shi and Tomasi, 1994] and Fluid simulation [Stam,

2003]. Figure 6.4 presents the Molen speed-up of the kernels and of the overall application with respect to the software. On average, the Molen system achieves a 2× improvement in execution time compared to the software. However, the performance of the Molen system is lower than the software in case of the *jpeg* application due to the high data communication. We measured that the ratio between data communication time and kernel processing time is up to 3.6× in this application. Therefore, optimal data communication is an essential demand to improve the system performance.

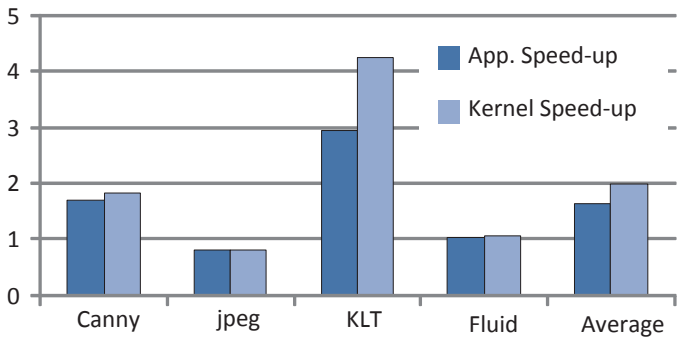


Figure 6.4: The speed-up of the baseline system compared to the software.

Table 6.2: Speed-up of the proposed system compared to software and the baseline system

App.	**#Func**[a]	**#Kernels**[b]	**w.r.t Software**		**w.r.t Baseline**	
			Application	Kernels	Application	Kernels
Canny	4	5	3.15×	3.88×	1.83×	2.12×
jpeg	4	5	2.33×	2.5×	2.87×	3.08×
KLT	3	3	3.72×	6.58×	1.26×	1.55×
Fluid	5	5	1.66×	1.68×	1.59×	1.60×

App.: Application;
[a] The number of functions in the application accelerated by hardware kernels;
[b] The number of hardware kernels accelerating the computationally intensive functions of the application;

We then implement our approach for each application on the Molen system. Beside the PLB bus used as the communication infrastructure, the developed crossbar and the adapted NoC are also used as the hybrid interconnect. Table 6.2 shows the speed-ups of the overall application and of the kernels of the proposed system with respect to software in Column 4 and Column 5, respectively. Those speed-ups with respect to the baseline system are also shown in the table.

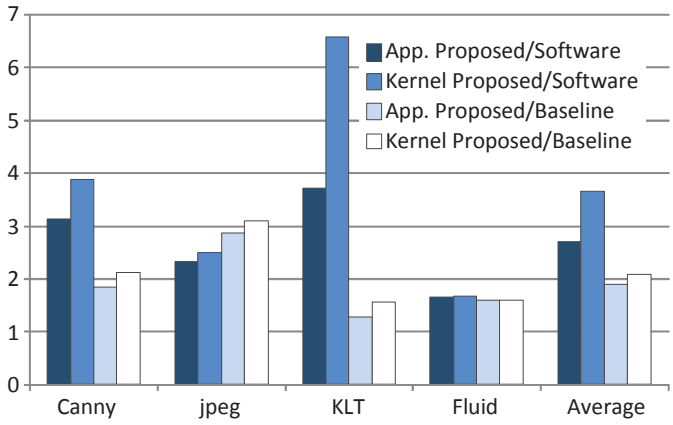


Figure 6.5: The overall application and the kernels speed-up of the proposed system compared to the software and baseline system.

Figure 6.5 compares the speed-up of the proposed system with respect to both software and the baseline. In each application, the first two chart bars illustrate the speed-ups of the overall application and of the kernels with respect to software while the last two chart bars show speed-ups with respect to the baseline system. As shown in the table as well as in the figure, when the proposed hybrid interconnect and parallelizing kernel processing are exploited, we achieve a speed-up of the overall application and of the kernels by up to 3.72× and 6.58× when compared to software, respectively (both in the KLT application). Compared to the baseline system, speed-ups of up to 2.87× for the overall application and 3.08× for the kernels are obtained (both in the case of the *jpeg* application).

RESOURCE UTILIZATION

Table 6.3 presents the hardware resource utilization for the whole system of the baseline, our system and the NoC-only system, in terms of the number of FPGA look-up tables (LUTs) and the number of FPGA registers. The NoC-only system is a system in which the parallel solution is applied, but only NoC is used for the interconnect of kernels (shared local memory solution is not used). Our adaptive mapping algorithm is also not applied in this system. The table also presents the components used for the hybrid interconnect in each application. As shown in the table, our system saves up to 33.1% LUTs and 30.2% registers compared to the NoC-only system. This result validates our goal that is to optimize the communication time while keeping the resources usage of the interconnect minimal. Without our strategy, the system is either the baseline (bus-based intercon-

nect) or NoC-only. The baseline system is a low performance system while the NoC-only system uses more hardware resources than our system. Meanwhile, our system (where both the shared memory solution and the NoC are used as interconnect for the kernels) achieves the same performance as the NoC-only system while using fewer resources. Figure 6.6 presents a comparison between resources used for interconnect and resources used for the kernels (computing) in our system normalized to the resources used for computing. As shown in the figure, the interconnect uses up to 40.7% resources compared to the resources used for computing at most.

Table 6.3: Hardware resource utilization comparison and the solution in the embedded system

App.	Resource	Baseline	NoC only	Proposed	Reduction[a]	Solution
Canny	LUT	9,926	17,894	15,227	14.9%	NoC, SM, P
	Register	12,707	21,059	18,657	11.4%	
jpeg	LUT	11,755	23,180	20,837	10.1%	NoC, SM, P
	Register	11,910	23,188	20,900	9.9%	
KLT	LUT	4,721	7,358	4,921	33.1%	SM
	Register	5,430	8,070	5,631	30.2%	
Fluid	LUT	19,125	24,552	24,156	1.6%	NoC
	Register	28,793	36,110	36,100	0.0%	

[a]Reduction to NoC-only system
App.: Application
SM: Shared local memory
P: Parallelism

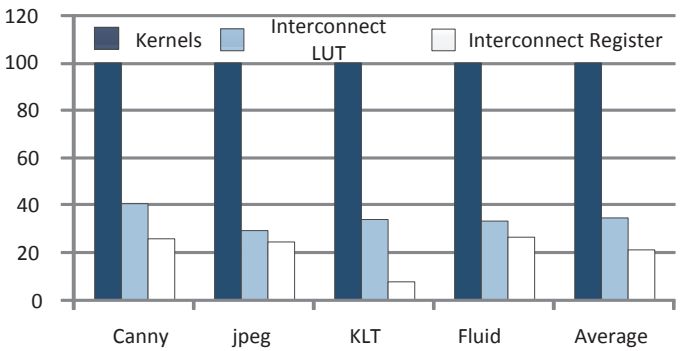


Figure 6.6: Interconnect resource usage normalized to the resource usage for the kernels

ENERGY CONSUMPTION

Figure 6.7 presents the comparison of the energy consumption between the baseline system and our proposed system normalized to the energy consumption of the baseline system. We use the Xilinx XPower Analyzer 13.2 tool to estimate the power consumption of each application in the two systems. The energy consumption is given by the product of the power consumption and the execution time. For both systems, the power consumption is almost identical, with a minor increase in our system (due to the increasing of resource usage for the hybrid interconnect). Therefore, our system consumes less energy per application due to the reduction in execution time. As shown in the figure, our system outperforms the baseline in all applications in terms of energy consumption. The maximum energy saved is 66.5% for the *jpeg* application.

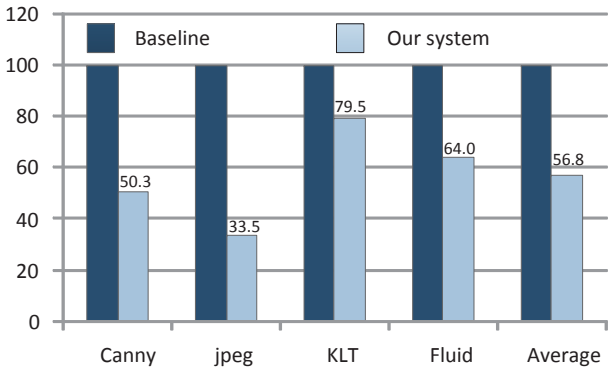


Figure 6.7: Energy consumption comparison between the baseline system and the system using custom interconnect with NoC normalized to the baseline system.

6.3.2. HIGH PERFORMANCE COMPUTING RESULTS

In this section, we present our experimental results in a high performance computing system - the Convey HC-2ex system [Convey Computer, 2012]. The Convey HC-2ex system consists of one Intel Xeon X5670 processor working at 2.93GHz and four Virtex-6 xc6vlx760 FPGA devices. The host processor and the accelerator kernels, mapped into the FPGA devices, can communicate through a Hybridcore Globally Shared Memory (HGSM) controlled by a Convey's HCMI.

Three applications are used in this experiment. Those are matrix multiplication, Canny edge detection and KLT feature tracker. We run all the applications on the host processor to get the software execution time. The accelerated functions are processed by all the 12 cores of the host processor using the OpenMP library, i.e. the host processor is fully utilized. The application is compiled by

GCC 4.2 with $-O2$ optimization level.

PERFORMANCE ANALYSIS

We develop the baseline system using the execution model presented in Section 3.2.1. However, it is not a fair comparison when the baseline system in high performance computing platform just has only one accelerator kernel for each function. Therefore, we apply the data parallelism solution for the baseline system. In other words, one function is accelerated by a number of kernels. The total number of accelerator kernels for both the baseline system and our proposed system is the same. Those accelerator kernels are mapped onto the FPGA devices. Because each accelerated function has a number of accelerator kernels, data input is divided into different segments. Each accelerator kernel processes one data segment. Vivado high level synthesis [Xilinx, 2014] generates the kernels for the functions from the ANSI C code. The whole system is synthesized with the Convey's scripts and Xilinx ISE 13.2 without any manual optimization.

In this system, the kernels (running at 150MHz) communicate together as well as communicate with the host through the HGSM. FPGA block RAMs are used as the local memories of the kernels. Segmented data is loaded from the HGSM to the local memories whenever the kernels are invoked. Segmented output of the kernels is written back to the HGSM right after the kernels finish. Figure 6.8 shows the speed-up of the baseline system for both the kernels and the overall application compared to the host processor where the accelerated functions are running on 12 cores at 2.93GHz. According to the figure, the KLT application does not have an improvement in performance in the accelerator system due to the very high data communication.

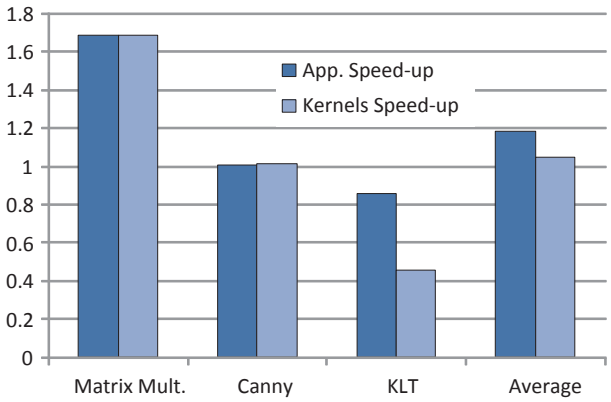


Figure 6.8: The speed-up of the baseline high performance computing system w.r.t software.

Table 6.4: High performance computing system results

App.	#Func[a]	#Kernels[b]	w.r.t Software		w.r.t Baseline	
			Application	Kernels	Application	Kernels
MM.[c]	2	128	2.61×	2.62×	1.54×	1.55×
Canny	4	64	1.55×	2.20×	1.53×	2.17×
KLT	3	56	1.02×	1.13×	1.20×	2.50×

App.: Application
[a]The number of functions in the application are accelerated by hardware kernels
[b]The number of hardware kernels accelerating the computationally intensive functions of the application
[c]Matrix Multiplication

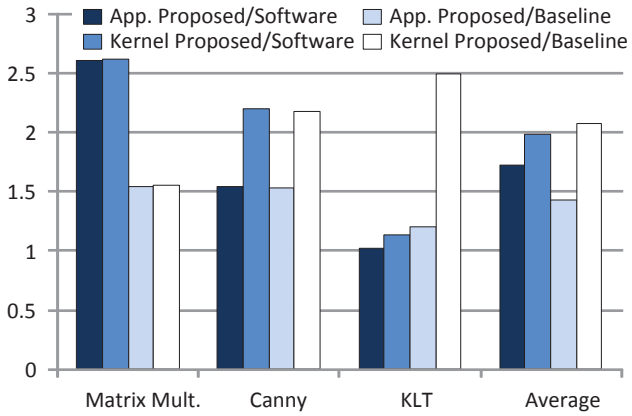


Figure 6.9: The overall application and the kernels speed-up of the proposed system compared to the software and baseline system.

Finally, we modify the baseline system using our proposed hybrid interconnect. Again, the developed 2×2 crossbar and the adapted NoC are used as the interconnect components for the kernels' communication. Table 6.4 presents in detail the speed-up of both the overall application and the kernels in the proposed system with respect to the software as well as the baseline system. The number of accelerated functions, as well as the number of accelerator kernels for the functions, are also shown in the table. Figure 6.9 depicts the speed-ups of the system using the proposed hybrid interconnect compared to both the software and the baseline system. As shown in the table as well as the figure, when the proposed hybrid interconnect is exploited, we achieve speed-ups of the overall application and of the kernels by up to 2.61× and 2.32×when compared to the

software version running on the 12 cores host processor, respectively (both in the case of the matrix multiplication application). Compared to the baseline system, the proposed hybrid interconnect system produces speed-ups of up to 1.54× for the overall application and up to 2.50× for the kernels.

RESOURCE UTILIZATION

Table 6.5 presents the hardware resource utilization for one FPGA device in the baseline system, our system and the NoC-only system in terms of the number of FPGA look-up tables (LUTs) and the number of FPGA registers. These values do not include the resource usage for Convey common components such as memory controllers, debug modules, etc. The table also presents the solutions used for the proposed system in each application. The NoC only has the same meaning as explained in the previous section. As shown in the table, our system saves up to 44.1% LUTs and 45.2% registers compared to the NoC-only system. This result, again, validates our goal that is to optimize the communication time while keeping the resources usage of the interconnect minimal. The goal is correct for both the embedded and the high performance computing platforms. Figure 6.10 presents a comparison between resources used for interconnect and resources used for the kernels (computing) in our system normalized to the resources used for computing. As shown in the figure, the interconnect and controller units use about 55% resources compared to the resources used for computing on average. However, in the case of the matrix multiplication application, the resource usage for the interconnect is higher than the resource usage for the computing kernels because the computing kernels in this application are quite simple.

Table 6.5: Hardware resource utilization comparison and the solution in the high performance system

App.	Resource	Baseline	NoC only	Proposed	Reduction[a]	Solution
MM	LUT	28,231	52,886	29,564	44.1%	SM, P
	Register	33,281	63,248	34,672	45.2%	
Canny	LUT	74,965	93,693	90,789	3.1%	NoC, SM, P
	Register	48,994	58,421	54,849	6.1%	
KLT	LUT	106,162	118,083	107,919	8.6%	SM, P
	Register	95,804	109,116	96,664	11.4%	

App.: Application; MM: Matrix Multiplication
[a]Reduction to NoC-only system
SM: Shared local memory
P: Parallelism

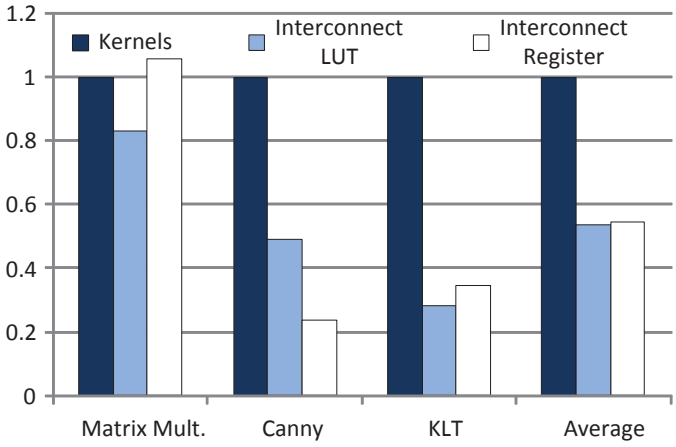


Figure 6.10: Interconnect resource usage normalized to the resource usage for the kernels.

ENERGY CONSUMPTION

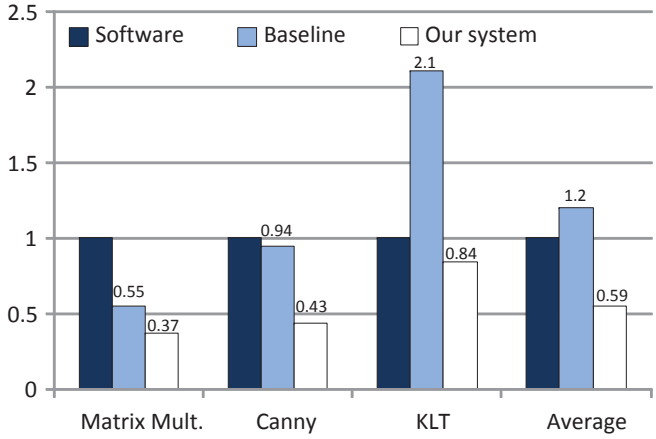


Figure 6.11: Energy consumption comparison between the baseline system and the system using custom interconnect with NoC normalized to the host processor energy consumption

In order to compare energy consumption for the kernels in the systems, we first calculate energy consumption of the software as the product of power consumption of the Xeon X5670 CPU (95W) and execution time. We then use Xilinx Power Analyzer 13.2 to estimate the power consumption of each FPGA device in both the baseline system and the proposed system. Since the Convey machine has four FPGAs, energy consumption for each application is approximated by four times the product of power consumption and execution time. Figure 6.11 shows the comparison of energy consumption of both the baseline system and

the proposed system normalized to the power consumption of the host processor. According the figure, our proposed system uses less energy than the host processor in all application while the baseline system uses more energy than the host processor in the case of the KLT application due to high execution time. We manage to save up to 63% energy consumption compared to the host processor in the case of matrix multiplication application compared to software.

6.3.3. MODEL COMPARISON

In this section, we compare the real execution time and the estimated value calculated using the models presented in Section 3.2.3 and Section 6.2.1. We analyze the Canny edge detection application in both the embedded system and the high performance computing system for this comparison. We first run the application on the baseline systems (embedded and high performance computing) to extract the computing time for accelerated functions, as well as the data communication time. Based on data communication profiling information, we calculate the total amount of data input and output of each accelerated kernel (the *UNDVS* value). Using that information, the average time for moving one byte of data through the communication infrastructure (θ) can be estimated.

With our proposed hybrid interconnect, data movement among the kernels can be hidden. Therefore, we can approximate the reduction in execution time by the equation Δ_c and Δ_n in Section 6.2.1 and Section 6.2.1. Finally, based on the computing time of the accelerator kernels, we estimate the reduction in time when the parallelism is exploited. We measure the actual execution time in different scenarios and compare the estimated value and the actual value. Those are the systems with the hybrid interconnect only and the systems with both the hybrid interconnect and parallelism.

Figure 6.12 depicts the QDU graph for the Canny application with an image 133×100 pixels used as data input which we use to do our experiment in the embedded platform (the image used for the high performance computing platform is 1024×1024 pixels). The graph in Figure 6.13(a) shows the comparison between the estimated reduction in time using the presented models and the actual value measured from the real execution on the corresponding platforms. The first and second columns of the figure compare the reduction in time of the embedded system when only our proposed hybrid interconnect is exploited (without parallelism). The third and fourth columns show the estimated reduction in time and actual reduction in time when parallelism is applied in the embedded system.

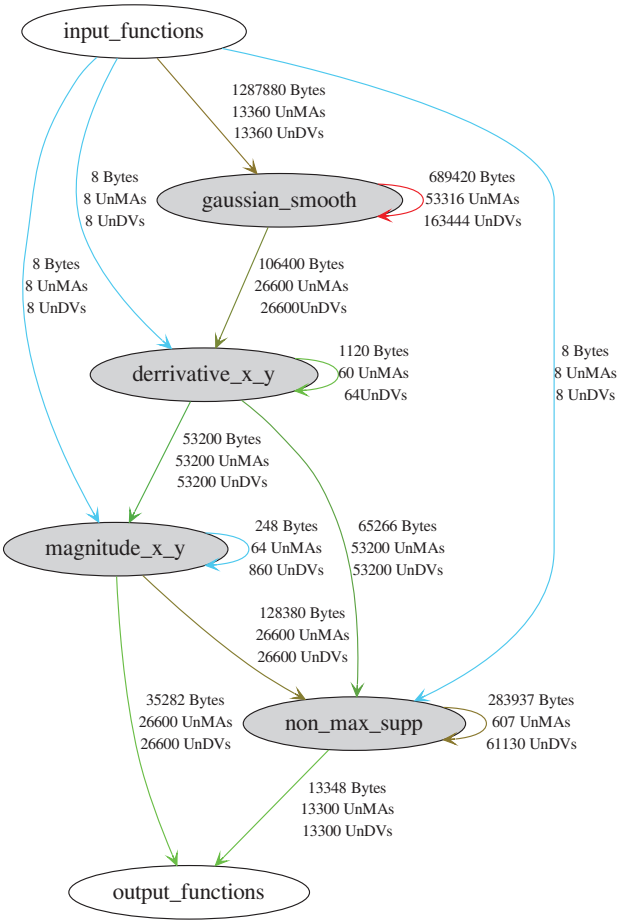
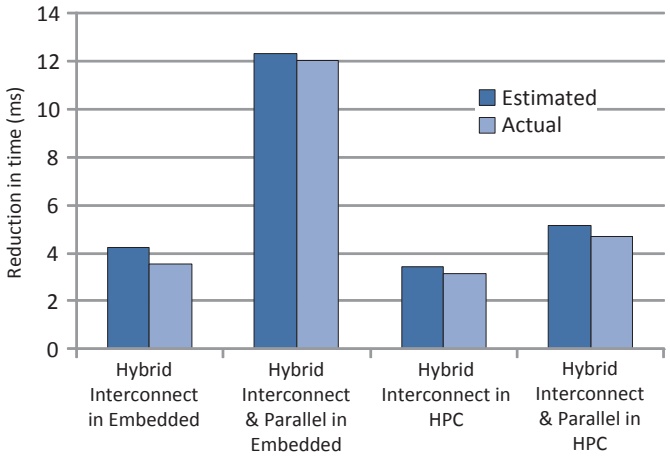


Figure 6.12: QDU graph for the canny application on the embedded platform.

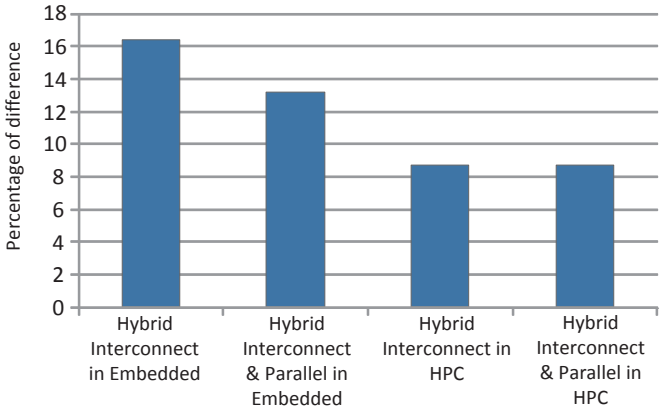
The rest four columns have the same meaning as the first four but for the high performance computing platform. The graph in Figure 6.13(b) shows the difference between the estimated value and real execution in percentage. This value is calculated by Equation 6.4:

$$Difference = \frac{estimated - actual}{estimated} \times 100. \tag{6.4}$$

As shown in the figure, our proposed estimated model is approaching the actual data. They are slightly different (16.4% at maximal) where the estimated model produces a larger reduction than real execution because there are some delays in system calls, context switching, etc.



(a)



(b)

Figure 6.13: The Comparison between estimated reduction in time and actual reduction time (a) in millisecond; (b) in percentage

6.4. SUMMARY

In this chapter, we presented an automated design approach to define an efficient custom interconnect for kernels in heterogeneous hardware accelerator systems using quantitative data communication profiling of applications. The hybrid interconnect includes a NoC, shared local memory solution, or both. For further optimization, parallelizing kernel processing was taken into consideration. We developed our experiments on both the Molen embedded platform and the Convey high performance computing platform. We compared our proposed systems with the original systems as well as the software running on the PowerPC at 400MHz in the Molen platform and on the 12 cores Intel Xeon processor in the Convey platform. The results showed that in both platforms, we achieved overall application speed-ups by up to 2.87× and by up to 1.55× compared to the baselines for the Molen platform and the Convey platform, respectively. Moreover, due to the reduction in execution time, our systems also used less energy compared to the baseline system by up to 66.5% and 63% for the embedded and high performance system, respectively.

Note. The content of this chapter is partially based on the following papers:

1. *C. Pham-Quoc, J. Heisswolf, S. Wenner, Z. Al-Ars, J.A. Becker, K.L.M. Bertels*, **Hybrid Interconnect Design for Heterogeneous Hardware Accelerators**, Design, Automation & Test in Europe Conference & Exhibition (DATE 2013), 18-22 March 2013, Grenoble, France

2. *C. Pham-Quoc, Z. Al-Ars, K.L.M. Bertels*, **Automated Hybrid Interconnect Design for FPGA Accelerators Using Data Communication Profiling**, 28th International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2014), 19-23 May 2014, Phoenix, USA

3. *C. Pham-Quoc, I. Ashraf, Z. Al-Ars, K.L.M. Bertels*, **Data Communication Driven Hybrid Interconnect Design for Heterogeneous Hardware Accelerator Systems**, (submitted), ACM Transactions on Reconfigurable Technology and Systems

7

ACCELERATOR ARCHITECTURE FOR STREAM PROCESSING

THIS chapter proposes a heterogeneous hardware accelerator architecture to support streaming image processing. Each image in a data-set is preprocessed on a host processor and sent to hardware kernels. The host processor and the hardware kernels process a stream of images in parallel. The Convey hybrid computing system is used to develop our proposed architecture. We use the Canny edge detection algorithm as our case study. The data-set used for our experiment contains 7200 images. Experimental results show that the system with the proposed architecture achieved a speed-up of the kernels by 2.13× and of the whole application by 2.40× with respect to a software implementation running on the host processor. Moreover, our proposed system achieves 55% energy reduction compared to a hardware accelerator system without streaming support.

7.1. INTRODUCTION

Stream processing is a computing paradigm in which a series of operations is applied to each element in a set of data. Stream processing not only can address the memory accessing bottlenecks by decoupled computation and memory accesses [Benjamin and Kaeli, 2007] but also can be used for applications in which data is continuously generated during the computation. Streaming image processing is one domain of stream processing. It is widely used in many application domains such as digital film processing [Bove and Watlington, 1995],

medical image diagnosis [Davis et al., 2007], real-time detection of changes in environmental phenomena [Rueda-Velasquez, 2007], video-based driver assistance [Claus and Stechele, 2010]. The computation in these systems is intensive especially when the resolution of data images is increased. A pure software implementation usually does not satisfy the real time performance requirement. Therefore, hardware acceleration for such systems is a necessity.

Meanwhile, with the rapid development of technology, more and more transistors can be integrated into one system. The more transistors a system has, the more power the system offers. However, we need to solve a couple of challenges such as power consumption, thermal emission, etc., when more transistors are integrated. Moreover, it is not easy and straightforward to develop all such above systems by using only hardware technologies such as FPGAs, ASICs, or integrated circuits. Hence, heterogeneous hardware acceleration represents one approach to overcome the challenges. A hardware accelerator system usually contains a host processor to execute some software functions of the application and some hardware fabric such as FPGA to accelerate some computationally intensive functions of the application. Another approach is homogeneous multicore systems. However, compared to homogeneous multicore systems, heterogeneous multicore systems offer more computation power and efficient energy consumption [Kumar et al., 2005].

In this chapter, we propose an architecture for a hardware accelerator system to support streaming image processing. In this architecture, streams are sequences of images which are processed by some software functions on the host processor and hardware kernels implemented on the hardware fabric. Our proposed system contains a host processor (a general purpose processor) and hardware kernels (accelerating computationally-intensive functions of the application) with controllers that support stream processing (in our experimental implementation, we use FPGA as hardware accelerator fabric). The host processor is responsible for receiving or sending the input or output image from or to other devices such as camera recorder or secondary hard disk, respectively. It also executes some functions of the application which cannot be executed on hardware. The hardware accelerator fabric consists of different kernels that implement the computationally intensive functions of the application. These kernels execute in parallel with the input streams. A shared memory is used for data communication of the host and the hardware kernels.

We implement our proposed architecture on the Convey hybrid computing

system HC-1 [Convey Computer, 2012] that contains an Intel Xeon 5408 CPU as the host processor and four Xilinx FPGAs as the hardware accelerator fabric. We use the Canny edge detection algorithm as our case study. A data-set containing 7200 images is used to test the performance of the proposed system. The experimental results show that our system achieve a speed-up of the overall application by 2.40× compared to a pure software implementation on the host processor and by 2.20× compared to a hardware accelerator system without streaming support.

The main contributions of the chapter are as follows: (1) propose a hardware accelerator architecture for streaming image processing; (2) present a speed-up estimation model for a streaming image processing application using a hardware accelerator system; (3) analyze the results of the synthesized implementation of the Canny edge detection algorithm, a well-known edge detection algorithm, in the proposed system.

The rest of the chapter is organized as follows. Section 7.2 introduces related work on hardware accelerator systems and streaming image processing and discusses the Canny edge detection algorithm. Section 7.3 presents our proposed architecture in detail. Section 7.4 illustrates how we implement the Canny edge detection algorithm on the proposed system. Our experimental results are shown and analyzed in Section 7.5. Finally, Section 7.6 concludes the chapter.

7.2. BACKGROUND AND RELATED WORK

In this section, we present the background and research related to hardware accelerator for streaming image processing. Beside the generic hardware accelerator systems in the literature presented in Section 2.3, here we summarize streaming image processing with hardware acceleration in the literature and introduce the Canny edge detection algorithm which we use as our case study.

7.2.1. STREAMING IMAGE PROCESSING WITH HARDWARE ACCELERATION

There are two approaches for streaming image processing. The first approach uses sequences of image-pixels as the streams while the second one uses sequences of images as the streams. The first approach introduces some overhead for segmenting the input image and combining the result after the processing. However, this approach does not require a large memory to store the image. The work in [Benderli et al., 2003] proposed an FPGA implementation of low latency 2-D wavelet transforms for streaming image processing in which a stream is a sequence of image-rows. The work in [Caarls et al., 2006] implemented an algo-

rithm skeleton to support streaming image processing on a heterogeneous platform containing an SIMD and an ILP processor.

The second approach of streaming image processing does not introduce the segmentation and combination overhead, but it requires a large memory to contain the images. The work in [Ha et al., 2012] proposed an Image-Set Processing streaming framework that uses the power of a heterogeneous CPU/GPU platform. In that work, the CPU is responsible for reading and writing an image while all image processing steps are done by the GPU. In contrast to that work, we use FPGA as accelerator and the host processor is responsible not only for reading and writing image but also for processing.

7.2.2. CANNY EDGE DETECTION ALGORITHM

Canny edge detection [Canny, 1986] is a well-know and powerful edge detection algorithm. In this work, we use the Canny edge detection algorithm as our case study. The program flow can be clearly partitioned into four main steps: (i) using the gaussian filter to remove noises (*gaussian* function); (ii) determining the edge strength (*derrivative_x_y* and *magnitude_x_y* functions); (iii) applying non-maximal suppression (*non_max_supp* function); and (iv) applying hysteresis (*hysteresis* function). The most computationally intensive function is *gaussian*. The size of a filter matrix used by *gaussian* affects the execution time of this function and the quality of the result. Figure 7.1 shows the results of the Canny algorithm with different size of the filter matrix.

(a) (b) (c)

Figure 7.1: (a) Original; (b) 6×6 filter matrix; (c) 3×3 filter matrix

7.3. ARCHITECTURE

This section presents the execution model of the host processor and hardware kernels, our proposed architecture, and our proposed multiple clock domains.

7.3.1. HARDWARE-SOFTWARE STREAMING MODEL

Figure 7.2 illustrates the hardware-software streaming model for a streaming image processing application using three hardware kernels named kernel_1, kernel_2, and kernel_3. In the rest of the chapter, we use the following terminology for our explanation:

- A *step* is a processing phase of the algorithm, e.g., the image processing algorithm in Figure 7.2 has five steps: an initializing step done by the host, three following steps done by kernel_1, kernel_2, and kernel_3 and a finalizing step done by the host;

- A *stage* is an execution phase in which one or more steps are executed in parallel on different input data, e.g., at stage 0 in Figure 7.2, kernel_1 processes image 1 while the rest of kernels are idle; at stage 1, kernel_1 processes image 2 and kernel_2 processes image 1 while kernel_3 is still idle.

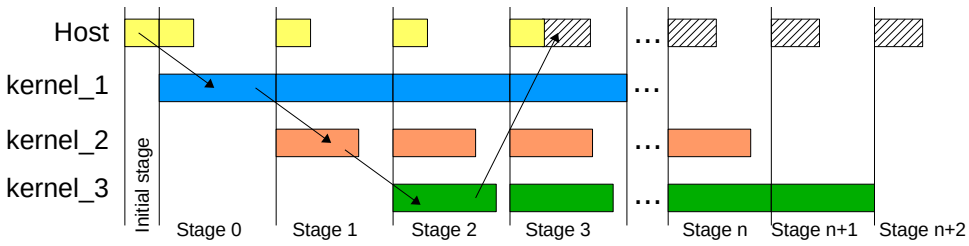


Figure 7.2: The streaming model

In this work, a shared double buffer mechanism is used for data communication between the host processor and the hardware kernels as well as among the hardware kernels. Assume that the data dependency between the host processor and hardware kernels follows the arrows in Figure 7.2 where the kernel at the top of an arrow consumes data produced by the kernel at the root of the arrow. For data communication between the host processor and kernel_1, we need two buffers (named $k1\_i1$ and $k1\_i2$ meaning that the input buffer 1 and 2 for kernel_1). During the initial stage, the host processor transfers the pre-processed

image 1 to $k1\_i1$. During stage 0, while kernel_1 processes data located on $k1\_i1$, the host processor pre-processes and transfers image 2 to buffer $k1\_i2$. During stage 1, while kernel_1 processes data located on $k1\_i2$, the host processor pre-processes and transfers image 3 to buffer $k1\_i1$. The same procedure is applied for the next stages. Similarly, we also need two buffers for each data communication between two hardware kernels such as $k2\_i1$ and $k2\_i2$ for data communication between kernel_1 and kernel_2. Two buffers named $k3\_o1$ and $k3\_o2$ are used for data communication between kernel_3 and the host processor. At stage 2, kernel_3 processes image 1 and writes its output to $k3\_o1$. During the next stage, kernel_3 writes the result of image 2 to $k3\_o2$ while the host processor does the post-processing for image 1 located in $k3\_o1$. When the first hardware kernel is started by a function call in the host processor (stage 0), the host processor sends the addresses of all the buffers to the corresponding kernels.

Due to the fact that we accelerate the most computationally intensive functions in hardware, the execution time of functions on the host is usually not longer than of the kernels. Assume that the execution time of kernel i for each image is t_i^k ($\forall i \in \{1,2,3\}$ - in the case presented in Figure 7.2) and of functions on the host for each image is t^s (we also assume that $t^s < min(t_i^k|_{i=1..3})$), the total execution time for the data-set in the case when stream processing is not applied (T_{nostr}) is as follows:

$$T_{nostr} = n \times (t^s + \sum_{i=1}^{3} t_i^k) \tag{7.1}$$

where n is the number of image in the data-set.

In the case when stream processing is applied as described above, the total execution time (T_{str}) for the data-set is as follows:

$$T_{str} = t_1^k + max(t_i^k|_{i=1,2}) + (n-2) \times max(t_i^k|_{i=1..3}) + max(t_i^k|_{i=2,3}) + t_3^k + t^s \tag{7.2}$$

Hence, the performance speed-up of stream processing compared to non-stream processing, S_p, is as follows:

$$S_p = \frac{T_{nostr}}{T_{str}} = \frac{n \times (t^s + \sum_{i=1}^{3} t_i^k)}{t_1^k + max(t_i^k|_{i=1,2}) + (n-2)max(t_i^k|_{i=1..3}) + max(t_i^k|_{i=2,3}) + t_3^k + t^s} \tag{7.3}$$

The speed-up (S_p) can be transformed as follows:

$$S_p < \frac{n \times (t^s + \sum_{i=1}^{3} t_i^k)}{n \times max(t_i^k|_{i=1..3})} < \frac{t^s + 3 \times max(t_i^k|_{i=\overline{1,3}})}{max(t_i^k|_{i=1..3})} \quad (7.4)$$

$$< \frac{t^s}{max(t_i^k|_{i=1..3})} + 3 \quad (7.5)$$

In general, the speed-up of a stream processing algorithm where m steps are accelerated by hardware compared to non-stream processing is lower than $(\frac{t^s}{max(t_i^k|_{i=1..m})} + m)$. Therefore, to improve the speed-up, we should reduce the execution time of the longest kernel and increase the number of accelerated steps.

7.3.2. SYSTEM ARCHITECTURE

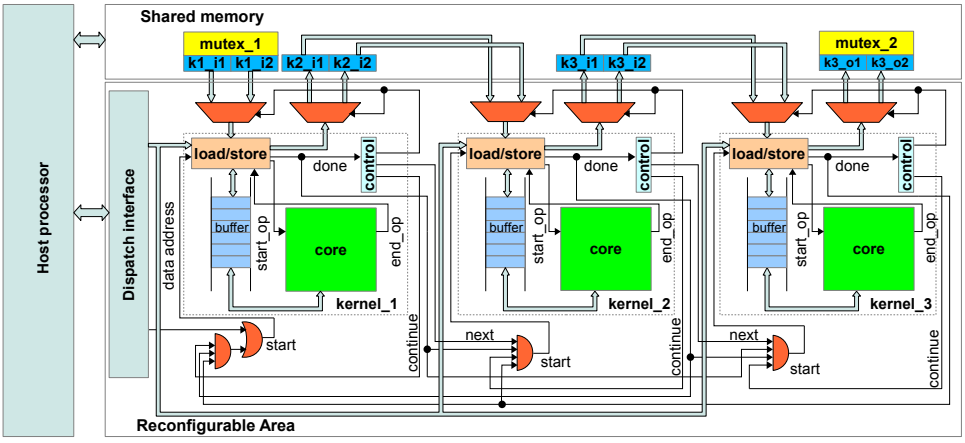


Figure 7.3: The system architecture supporting pipeline for streaming applications

Figure 7.3 depicts our proposed architecture with three hardware kernels representing three kernel types: *input kernel* (kernel_1), *intermediate kernel* (kernel_2), and *output kernel* (kernel_3). The input kernel receives data from the host and processes the first accelerated step while the output kernel processes the final accelerated step and transfers the results of the kernels to the host. The intermediate kernel is responsible for the intermediate accelerated step. An image processing algorithm may require one or more intermediate accelerated steps corresponded to one or more intermediate kernels.

The shared buffers in the shared memory such as $k1\_i1$, $k2\_i1$ are explained

in the previous section. Due to the shared memory, data does not need to be transferred from buffer to buffer. At each kernel, the *control* unit selects the right buffer for the current stage based on the number of images the kernel processed. Based on the number of images of the data-set, the control unit also determines if the kernel should continue executing or not. The host sends the addresses of the shared buffers and the number of images of the data-set to the kernels. This information is transferred through the *dispatch interface* unit.

The *core* unit in each kernel is responsible for the main task of the kernel. The core units process data stored in their local *buffers*, usually the on-chip memory. The *load/store* unit in each kernel is responsible for loading data from the corresponding shared buffer (the shared memory) to the local buffer when the kernel is started. The *load/store* unit writes the result from the local buffer to the corresponding shared buffer when the core finishes (*end_op* signal is asserted). The core unit is started (*start_op* signal is asserted) when input data is ready.

The kernel is launched whenever its *start* signal is asserted. The starting of the kernels is synchronous with each other. A kernel is going to be started if other kernels which are responsible for the previous steps, are executed before. In other words, the kernel can be started if and only if all the *done* signals (the signal used for synchronization) of all the kernels are active and input data for the kernel is ready. The *done* signal is active if no workload needs to be done by the corresponding kernel. For example, in the first stage, kernel_1 (in Figure 7.3) is invoked by the *start* signal from the dispatch interface (function call from the host) while the other kernels are idle because their input data is not yet ready (their *done* signals are active). When kernel_1 finishes, its *done* signal and *next* signal are asserted to notify kernel_2 that input data is ready. During the next stage, kernel_1 and kernel_2 process the next image and the output of kernel_1, respectively, while kernel_3 is not executed due to the de-asserted *next* signal of kernel_2.

7.3.3. MULTIPLE CLOCK DOMAINS

Because we separate the shared memory access operation and the computational operation, we can use different clock domains for the system. The load/store unit, control unit and dispatch interface use a default system clock frequency, which is usually at a moderate level. Meanwhile, the core can execute with a higher clock frequency to improve the performance. Assume that the system clock frequency is f_{sys} and the clock frequency for the core is f_{core}. The speed-

up estimation in Equation 7.3 is modified as follow:

$$S'_p \approx \frac{n(t^s + \sum_{i=1}^{3} t_i^k)}{[t_1^k + max(\, t_i^k|_{i=1,2}) + (n-2)max(\, t_i^k|_{i=1..3}) + max(\, t_i^k|_{i=2,3}) + t_3^k]\frac{f_{sys}}{f_{core}} + t^s} \tag{7.6}$$

Compared to Equation 7.3, we have $S'_p > S_p$ due to the assumption $f_{core} > f_{sys}$.

7.4. CASE STUDY: CANNY EDGE DETECTION

This section presents the implementation of the Canny edge detection algorithm (presented in Section 7.2.2) using our proposed architecture. We use the ANSI C implementation version provided by the University of South Florida [Florida, 1999] in our experiment.

The *gprof* profiling tool [Graham et al., 1982] is used to identify the most computationally intensive functions. We accelerate four functions of the Canny application by hardware kernels. Those are *gaussian*, *derrivative_x_y*, *magnitude_x_y* and *non_max_supp*. The DWARV compiler [Nane et al., 2012] is used to generate a VHDL description for those functions (the core units in our architecture). We simulate those kernels to estimate their execution time by ModelSim. The simulation result shows that the most computationally intensive function, *gaussian*, takes about 2.2× longer than the total execution time of the three other kernels, *derrivative_x_y*, *magnitude_x_y*, and *non_max_supp*. Therefore, we decide to implement two hardware kernels for the *gaussian* function, i.e., two images are processed by the *gaussian* kernels at each stage. Moreover, the kernels of the *magnitude_x_y* and *non_max_supp* functions can be started right after the finishing of *derrivative_x_y* kernel because their input data is ready right after the *derrivative_x_y* kernel finishes. Therefore, we modify the proposed architecture such that the three kernels *derrivative_x_y*, *magnitude_x_y* and *non_max_supp* can be started asynchronously with the *gaussian* kernel as depicted in Figure 7.4. During a stage (except Stage 0, Stage n and Stage $n+1$), while two *gaussian* kernels process two images (k and $k+1$ where $2 < k < 2n-1$; $2n$ is the number of images of the data-set), the rest three kernels are executed two times to process the results of the gaussian kernels in the previous stage (the image $k-2$ and $k-1$).

We use the Convey hybrid computing system HC-1 [Convey Computer, 2012] as a hardware accelerator platform to develop our architecture. The Convey system consists of one Intel Xeon 5408 CPU, working at 2.14GHz, as host processor

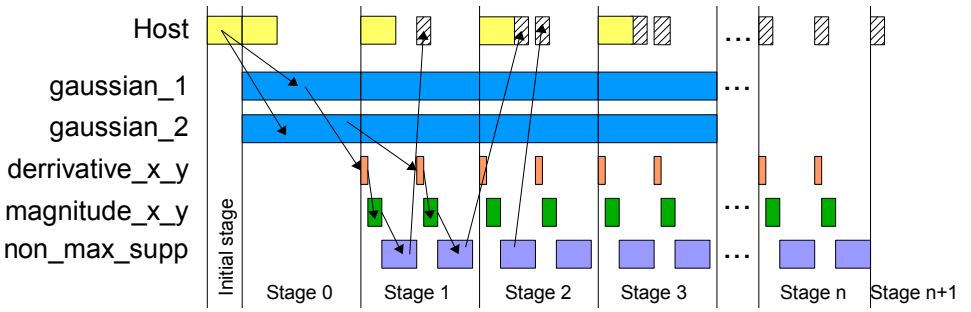


Figure 7.4: The execution model and data dependency between kernels for the Canny algorithm

and four Xilinx Virtex 5 LX330 FPGA as the hardware accelerator fabric (so-called co-processor). Figure 7.5 depicts the Convey system architecture. The communication between the host and the co-processors is done by the HCMI bus. The shared memory consists of 128GB for the host processor and 64GB for the co-processors.

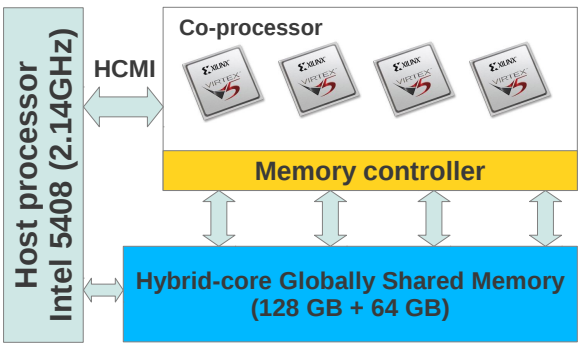


Figure 7.5: The Convey hybrid computing system

In this case study, we configure each FPGA device with two *gaussian* kernels, one *derrivative_x_y* kernel, one *magnitude_x_y* kernel and one *non_max_supp* kernel. Therefore, we can process 8 images at one stage following the execution model in Figure 7.4. FPGA devices run independent from each other. The kernels in Device 0 process image i and $(i+1)$ in data set where i is multiple of 8. Consequently, the kernels in Device 1 process image $(i+2)$ and $(i+3)$; the kernels in Device 2 process image $i+4$ and $i+5$; and the kernels in Device 3 process image $i+6$ and $i+7$.

7.5. EXPERIMENTAL RESULT

This section presents our experimental results with the Canny application using a data-set containing 7200 images extracted from a 5 minutes video. The image size is 152 × 114 pixels. We set the standard deviation gaussian, low and high thresholds to 2.0, 0.5 and 0.5, respectively (i.e., we use an 11 × 11 filter matrix for the *guassian* function). Beside the system presented in the previous section, we develop two other accelerator systems for a comparison. Firstly, we run the software version on the host processor (Intel Xeon 5408 Quad-core working at 2.14GHz). Secondly, we develop a hardware accelerator system for the application without streaming but with multiple clock domains, i.e., following the non-streaming model presented in Section 7.3.1 and using f_{core} for the cores while f_{sys} for the rest units in the kernels. We, then, build another hardware accelerator system with streaming model but all units use the system clock frequency (f_{sys}). Finally, we implement the system with streaming model and multiple clock domains (the proposed architecture). All the systems have two hardware kernels for the *gaussian* function and one hardware kernel for each another function. In the multiple clock domains systems, the cores of the kernels are executed 225MHz (f_{core}) while other units run at 150MHz (f_{sys}).

Table 7.1 shows the execution time of the application with different systems and the speed-up compared to the pure software execution (System 1). The streaming system with multiple clock domains achieves a speed-up of the overall application by 2.40× compared to the software execution and by 2.20× compared to the non-streaming system (System 2). Moreover, the proposed system achieves a speed-up of the overall application by 1.47× compared to the streaming system without multiple clock domains (System 3).

In System 3 and System 4, the speed-up of the kernels are lower than the speed-up of the whole application (1.45× compared to 1.63× and 2.13× compared to 2.40×, respectively) due to the fact that we execute software hardware streaming model. In other words, the software task for $n-1$ image is done in parallel with the hardware tasks (the kernels). The execution time of the whole application is the sum of the execution time of hardware kernels, software initializing step for image 1 and finalizing step for image n. Therefore, the execution time for the whole application is slightly longer than the kernels. Figure 7.6 shows the speed-up of both kernels and the whole application of the three last systems with respect to the first system - the software implementation only.

Table 7.2 shows the resource usage of the kernels in the streaming system

Table 7.1: Application execution time and speed-up of different systems

System	Kernel time	App. time	Kernel speed-up	App. speed-up
System 1	46.88s	52.85s	1.0×	1.0×
System 2	40.46s	48.43s	1.16×	1.09×
System 3	32.44s	32.46s	1.45×	1.63×
System 4	21.97s	21.99s	2.13×	2.40×

App.: Application
System 1: Software only
System 2: Non-streaming model with multiple clock domains
System 3: Streaming with the same clock frequency for all units
System 4: Streaming with multiple clock domains (the proposed architecture)

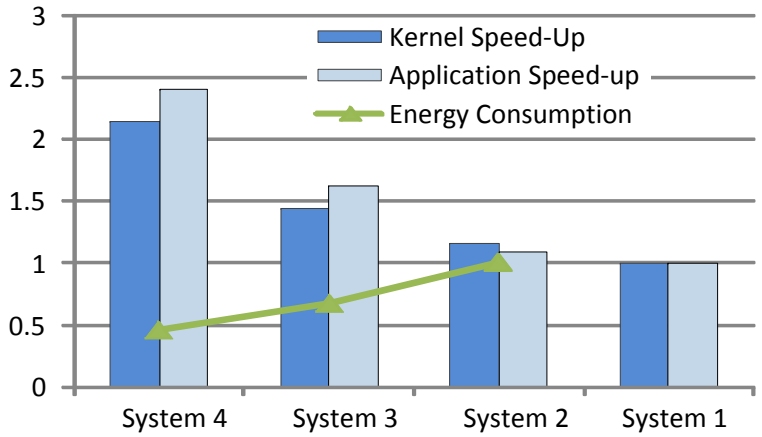


Figure 7.6: The speed-up and energy consumption comparison between the systems

with multiple clock domains (the proposed architecture) in term of the number of LUTs, the number of registers, and the number of DSPs for each kernel (including the core, the load/store unit and the control unit). It also presents the total resource usage for the whole system (including other modules used by the Convey system such as the memory controller and the dispatch interface). Each kernel has 132KB buffer implemented by block RAM (11.45% FPGA BRAM). The number of block RAMs used for the whole system is 904KB (78%).

We use Xilinx XPower Analyzer to estimate the power consumption for each FPGA device. The total power consumption of the hardware fabric in the three systems is almost identical. Table 7.3 shows the power consumption distribution

Table 7.2: The resource usage for each kernel and the whole streaming system with multiple clock domains

Kernel	#LUTs	#Registers	#DSP
gaussian	10,231 (5%)	13,438 (6.3%)	7 (3.6%)
derrivative_x_y	2,907 (1%)	2,246 (1%)	0
magnitude_x_y	4,058 (1%)	3650 (1%)	5 (2%)
non_max_supp	8,571 (4.1%)	10,020 (4.9%)	5 (2%)
System	104,534 (50.4%)	113,781 (54%)	24 (12%)

Xilinx Virtex 5 LX330 contains 207,360 LUTs, 207,360 Register, 192 DSP and 1152 KB block RAM

of one FPGA device as well as the amount of resource usage (LUTs and FFs) for each system. The amount of resource usage for all three systems is almost identical. However, the power consumption of each resource type (clock, logic and BRAM) of System 3 is smallest because System 3 uses clock frequency at 150MHz for all component while the two other systems use a 225MHz clock frequency for some components. The power consumption of System 4 is larger than of System 2 because System 4 uses more logic elements (LUTs and FFs) than System 2. These logic elements are used for the streaming controllers.

We compute the energy consumption of the hardware fabric (without energy consumed by the host processor) by the product of power consumption and the execution time. Due to the short execution time, the system using streaming model with multiple clock domains (System 4) uses less energy than the non-streaming model system (System 2) and the streaming model system without multiple clock domains (System 3). Figure 7.6 shows the energy consumption normalized to energy consumption of the system with non-streaming and multiple clock domains. As shown in the figure, System 4 achieves 55% energy reduction compared to System 2.

7.6. SUMMARY

This chapter presented a heterogeneous hardware accelerator architecture with multiple clock domains for a streaming image processing application. The work also introduced a model to estimate the speed-up of a streaming image processing system compared to a non-streaming one. The Canny edge detection algorithm was used as a case study. Experimental results show that the streaming system achieved a speed-up of the overall system by 2.40× and of the kernels

Table 7.3: Power consumption (W) and resource usage of the systems

	Resource	System 2	System 3	System 4
Power	Clock	0.501	0.383	0.502
	Logic	0.695	0.798	0.827
	BRAM	1.721	1.456	1.721
	System	**18.334**	**18.038**	**18.481**
Amount	FF	112,384	113,781	
	LUT	102,836	104,534	

The systems are explained in Table 7.1

only by 2.13× compared to the software implementation executed on Intel Xeon 5408 CPU. The experiment also compared the proposed system to the streaming system without multiple clock domains. The streaming system with multiple clock domains consumes less energy than the stream system without multiple clock domains although the system without multiple clock domains has a lower power consumption.

Note. The content of this chapter is based on the following paper:

C. Pham-Quoc, Z. Al-Ars, K.L.M. Bertels, **Heterogeneous Hardware Accelerator Architecture for Streaming Image Processing**, International Conference on Advanced Technologies for Communications (ATC 2013), 16-18 October 2013, Ho Chi Minh City, Vietnam

8

CONCLUSIONS AND FUTURE WORK

IN this dissertation, we addressed the challenges related to hybrid interconnect design for heterogeneous hardware accelerator systems. In this chapter, we summarize the work presented in this dissertation. We also emphasize our contributions in this chapter. Finally, we present the future work to extend our current work.

8.1. SUMMARY

The work presented in this dissertation is divided into the following chapters:

Chapter 1 introduces an overview of the challenges that we address in this dissertation. Five different research questions are presented in this chapter, in addition to a list of our contribution.

Chapter 2 shows the background and the related work. In this chapter, we present the on-chip interconnects in the literature as well as five different taxonomies to classify the on-chip interconnects. Advantages and disadvantages of some well-known interconnects are also analyzed. A survey on the hybrid interconnects in the literature is given, where we classify hybrid interconnects in the literature into two categories: *mixed topologies hybrid interconnect* and *mixed architecture hybrid interconnect*. This chapter also presents the state-of-the-art hardware accelerator systems in both academia and industry; and we zoom in on their interconnect aspects. Finally, data communication techniques in the literature for such systems are also discussed.

In Chapter 3, an overview of our approach using data communication profiling to define a custom interconnect for a specific application is presented. We also propose a data communication driven quantitative execution model. Finally, to further improve the system performance, parallelizing kernel processing is also analyzed.

Chapter 4 analyzes different alternative interconnection solutions including direct memory access (DMA), crossbar, a combination of DMA and crossbar, and NoC to improve system performance of a bus-based hardware accelerator. In this chapter, we also propose the analytical models to predict performance for these solutions and implement them in practices. We profile the application to extract data input for the analytical models. A comparison between the analytical models and real execution is done to validate the analytical models. Experimental results show that the proposed analytical models match closely the measured data from execution using an embedded platform.

In Chapter 5, a heuristic-based approach to design an application specific hardware accelerator systems with a custom interconnect using quantitative data communication profiling is proposed. A number of solutions are considered to define the most optimized system in term of system performance. Those are crossbar-based shared local memory, DMA support parallel computing, local buffer, and hardware duplication. Experimental results with different applications are analyzed to validate the proposed heuristic approach as well as the contribution of each solution.

Chapter 6 proposes an automated hybrid interconnect design strategy to create an efficient custom interconnect for accelerator kernels in a hardware accelerator system. The aim of the hybrid interconnect is to improve system performance by reducing data communication overhead. Our custom interconnect includes a NoC, shared local memory solution, or both. Depending on the quantitative data communication profiling of each application, the interconnect for the application is built using the automated strategy. An adaptive data communication-based mapping is proposed to map the computing cores to the system interconnect to obtain the most optimized hardware resource usage. Experiments on both an embedded hardware accelerator system as well as a high performance computing platform are performed to validate the proposed automated design strategy.

In Chapter 7, we present a case study of a heterogeneous hardware accelerator architecture supporting streaming image processing. Each image in a data-

set is pre-processed by the host processor and sent to the hardware kernels. The host processor and the hardware kernels process a stream of images in parallel. Convey hybrid computing system is used to implement the proposed architecture. The Canny edge detection is considered as our case study in this chapter.

8.2. CONTRIBUTIONS

As already discussed in Chapter 1, interconnect in a multicore system plays an important role in system design. During this work, we focus on designing a hybrid interconnect that helps the systems improve performance while keeping the hardware resource overhead minimal. Figure 8.1 shows that hybrid interconnect can achieve high performance, scalability, and area-efficiency when compared to standard interconnects. This dissertation has three different contributions: 1) Modeling, 2) Automation, 3) and Demonstration. The detailed contributions of this dissertation are summarized as follows.

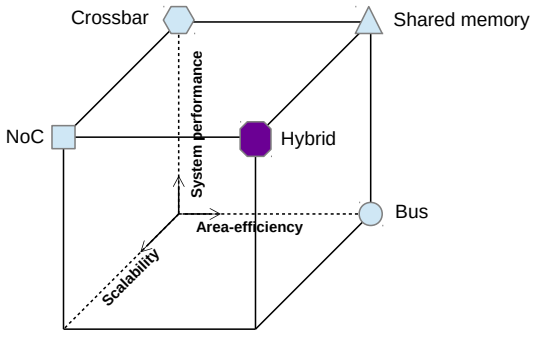


Figure 8.1: Interconnects comparison.

Contribution 1 *Modeling*

We introduce an efficient execution model, which takes data communication profiling into account, for a heterogeneous hardware accelerator system. Based on a detailed and quantitative data communication profiling, an accelerator kernel knows exactly which subsequent kernels will consume its data output. Therefore, instead of sending data output back to the main memory, the kernel can deliver its data output to the consuming kernels through our proposed application specific hybrid interconnect in parallel with its execution. The consuming kernels, then, do not need to wait for data movement to complete. Consequently, system performance is improved because data communication time is alleviated.

Contribution 2 *Automation*

We proposed a heuristic approach taking data communication patterns inside an application into consideration to design a hardware accelerator system for the application with an optimized custom interconnect. The main goal of this approach is to achieve the most optimized system performance. The approach mainly focuses on data communication optimization since this is a primary anticipated bottleneck for system performance. A number of solutions are considered in this heuristic approach. Those are crossbar-based shared local memory, DMA, local buffer, and hardware duplication. An analytical model is proposed to estimate system performance improvement with the solutions. The approach is mainly useful for embedded systems.

We propose an automated approach using a detailed and quantitative data communication profiling to define a hybrid interconnect for each specific application, resulting in the most optimized performance with a low hardware resource usage and energy consumption. Evidently, kernels and their communication patterns are different from one application to the other. Each application should have a specific interconnect to get data efficiently to the kernels that need it. Therefore, we propose this automated hybrid interconnect design approach. We call it hybrid interconnect because ultimately the entire interconnect will consist of not only a NoC but also uni- or bi-directional communication channel or shared local memory for data exchanges between the kernels. The design approach results in an optimized hybrid interconnect in term of system performance while keeping the hardware resource usage for the interconnect minimal.

Contribution 3 *Demonstration*

We demonstrate our proposed approach in both an embedded platform and a high performance computing platform, to validate the benefits of the hybrid interconnect. The results in both platforms, the Molen architecture implemented on a Xilinx ML510 board and the Convey high performance computing system, show that the hybrid interconnect improves system performance and reduces energy consumption compared to the systems without the hybrid interconnect.

8.3. FUTURE WORK

In this dissertation, we addressed the challenges related to interconnect design to improve system performance because interconnect plays an important role in multicore systems. We identify five followup open issues for future research.

First, runtime reconfigurability is an issue to be considered. An application can change dynamically while running. Depending on user input, application can be in different processing modes. Therefore, runtime reconfigurability can be used such that each application can deploy its best interconnect infrastructure leading to faster execution and less overall energy consumption. However, overheads such as performance overhead incurred by reconfiguration should be taken into account. This approach is beneficial if system performance improvement by the hybrid interconnect can compensate the reconfiguration overhead.

System/architecture-based approach is another future topic in this research direction. In this work, we focus on heterogeneous hardware accelerator systems in which accelerator kernels are usually dedicated circuits. However, future accelerator systems may consist of some other processing types such as VLIW processors or DSP processors. Moreover, the memory hierarchy may be different when those types of processing cores are used. Therefore, a more generic approach that also takes the system architecture into consideration is still an open issue.

When VLIW or DPS processors are used as accelerators instead of dedicated circuits, bringing computation to data is an interesting topic. Those processors usually have two local memory types: data cache and instruction cache. In contemporary approaches, we usually move output data from one computing core to other computing cores. However, the amount of data that needs to be processed is continuously growing while the instructions for processing data do not often change. Therefore, instead of developing the hybrid interconnect for data communication between the cores, we can design the interconnect to transfer instructions from core to core while keeping data local.

In this work, different tools including profiling tools such as gprof and QUAD, high level synthesis tools such as DWARV and Vivado, and system design tools such as ISE are used. However, they are used separately and called manually. Therefore, a fully automated design framework, that links these tools automatically together, is an open issue. In such a proposed framework, all the steps presented in Section 3.1.2 would be performed automatically without any interaction with designers.

Finally, merging and partitioning functions are interesting future research approaches. In this work, we are considering to choose the most computationally intensive functions that can be accelerated by the hardware fabric first. We, then, design the hybrid interconnect for those accelerators. However, in some

cases, there exists a large data communication between some non-computationally intensive functions, that are executed by the host, and the accelerator kernels. This large amount of data is transferred from the host to the kernels and vice versa through the system communication infrastructure. This data movement significantly affects system performance. Therefore, considering to merge these functions to the computationally intensive functions to reduce data communication is one solution that should be taken into consideration. Besides, a complicated dedicated circuit may require a low frequency. This can lead reducing the system frequency to satisfy timing issues. Therefore, considering to partition a complicated function to simpler functions before designing the hybrid interconnect is also an interesting future work.

BIBLIOGRAPHY

L. Acasandrei and A. Barriga. 2013. AMBA bus hardware accelerator IP for Viola-Jones face detection. *Computers Digital Techniques, IET* 7, 5 (September 2013), 200–209. DOI:http://dx.doi.org/10.1049/iet-cdt.2012.0118

Altera. 2008. Accelerating Nios II Systems with the C2H Compiler Tutorial. (2008).

ARM Limited. 1999. AMBA Specification (Rev 2.0). (1999).

ARM Limited. 2001. Multi-layer AHB overview. (2001).

Imran Ashraf, S. Arash Ostadzadeh, Roel Meeuws, and Koen Bertels. 2012. Communication-aware HW/SW co-design for heterogeneous multicore platforms. In *Proceedings of the 2012 Workshop on Dynamic Analysis (WODA 2012)*. ACM, New York, NY, USA, 36–41. DOI:http://dx.doi.org/10.1145/2338966.2336806

A. Avakian, J. Nafziger, A. Panda, and R. Vemuri. 2010. A reconfigurable architecture for multicore systems. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. 1–8. DOI:http://dx.doi.org/10.1109/IPDPSW.2010.5470753

James Balfour and William J. Dally. 2006. Design Tradeoffs for Tiled CMP On-chip Networks. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*. ACM, New York, NY, USA, 187–198. DOI:http://dx.doi.org/10.1145/1183401.1183430

A.O. Balkan, Gang Qu, and U. Vishkin. 2006. A Mesh-of-Trees Interconnection Network for Single-Chip Parallel Processing. In *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on*. 73–80. DOI:http://dx.doi.org/10.1109/ASAP.2006.6

Aydin O. Balkan, Gang Qu, and Uzi Vishkin. 2008. An Area-efficient High-throughput Hybrid Interconnection Network for Single-chip Parallel Processing. In *Proceedings of the 45th Annual Design Automation Conference (DAC '08)*. ACM, New York, NY, USA, 435–440. DOI:http://dx.doi.org/10.1145/1391469.1391583

J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka. 2007. Dynamic and Partial FPGA Exploitation. *Proc. IEEE* 95, 2 (Feb 2007), 438–452. DOI:http://dx.doi.org/10.1109/JPROC.2006.888404

O. Benderli, Y.C. Tekmen, and N. Ismailoglu. 2003. A real-time, low latency, FPGA implementation of the 2-D discrete wavelet transformation for streaming image applications. In *Digital System Design, 2003. Proceedings. Euromicro Symposium on*. 384–389. DOI:http://dx.doi.org/10.1109/DSD.2003.1231971

L. Benini and G. De Micheli. 2002. Networks on chips: a new SoC paradigm. *Computer* 35, 1 (Jan 2002), 70–78. DOI:http://dx.doi.org/10.1109/2.976921

L. Benini, E. Flamand, D. Fuin, and D. Melpignano. 2012. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*. 983–987. DOI:http://dx.doi.org/10.1109/DATE.2012.6176639

MichaelG. Benjamin and David Kaeli. 2007. Stream Image Processing on a Dual-Core Embedded System. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Stamatis Vassiliadis, Mladen Bereković, and TimoD. Hämäläinen (Eds.). Lecture Notes in Computer Science, Vol. 4599. Springer Berlin Heidelberg, 149–158. DOI:http://dx.doi.org/10.1007/978-3-540-73625-7_17

B. Betkaoui, D.B. Thomas, W. Luk, and N. Przulj. 2011. A framework for FPGA acceleration of large graph problems: Graphlet counting case study. In *FPT*. 1–8. DOI:http://dx.doi.org/10.1109/FPT.2011.6132667

C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen. 2005. DyNoC: A dynamic infrastructure for communication in dynamically reconfugurable devices. In *Field Programmable Logic and Applications, 2005. International Conference on*. 153–158. DOI:http://dx.doi.org/10.1109/FPL.2005.1515715

Shekhar Borkar and Andrew A. Chien. 2011. The Future of Microprocessors. *Commun. ACM* 54, 5 (May 2011), 67–77. DOI:http://dx.doi.org/10.1145/1941487.1941507

Stephan Bourduas and Zeljko Zilic. 2011. Modeling and evaluation of ring-based interconnects for Network-on-Chip. *Journal of Systems Architecture* 57, 1 (2011), 39 – 60. DOI:http://dx.doi.org/10.1016/j.sysarc.2010.07.002 Special Issue On-Chip Parallel And Network-Based Systems.

V. M. Bove, Jr. and J. A. Watlington. 1995. Cheops: A Reconfigurable Data-flow System for Video Processing. *IEEE Trans. Cir. and Sys. for Video Technol.* 5, 2 (April 1995), 140–149. DOI:http://dx.doi.org/10.1109/76.388062

J.D. Brown, S. Woodward, B.M. Bass, and C.L. Johnson. 2011. IBM Power Edge of Network Processor: A Wire-Speed System on a Chip. *Micro, IEEE* 31, 2 (March 2011), 76–85. DOI:http://dx.doi.org/10.1109/MM.2011.3

W. Caarls, P.P. Jonker, and H. Corporaal. 2006. Algorithmic skeletons for stream programming in embedded heterogeneous parallel image processing applications. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. 9 pp.–. DOI:http://dx.doi.org/10.1109/IPDPS.2006.1639351

J. Camacho, J. Flich, A. Roca, and J. Duato. 2011. PC-Mesh: A Dynamic Parallel Concentrated Mesh. In *Parallel Processing (ICPP), 2011 International Conference on*. 642–651. DOI:http://dx.doi.org/10.1109/ICPP.2011.21

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst.* 13, 2, Article 24 (Sept. 2013), 27 pages. DOI:http://dx.doi.org/10.1145/2514740

John Canny. 1986. A Computational Approach to Edge Detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* PAMI-8, 6 (Nov 1986), 679–698. DOI:http://dx.doi.org/10.1109/TPAMI.1986.4767851

Jongsok Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski. 2012. Impact of Cache Architecture and Interface on Performance and Area of FPGA-Based Processor/Parallel-Accelerator Systems. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. 17–24. DOI:http://dx.doi.org/10.1109/FCCM.2012.13

Ken Christensen. 2012. Christensen Tool page - Department of Computer Science and Engineering - University of South Florida. (2012). http://www.csee.usf.edu/~christen/tools/toolpage.html#bloom

Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: An In-fabric Memory Architecture for FPGA-based Computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. ACM, New York, NY, USA, 97–106. DOI:http://dx.doi.org/10.1145/1950413.1950435

Eric S. Chung, Michael K. Papamichael, Gabriel Weisz, James C. Hoe, and Ken Mai. 2012. Prototype and Evaluation of the CoRAM Memory Architecture for FPGA-based Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '12)*. ACM, New York, NY, USA, 139–142. DOI:http://dx.doi.org/10.1145/2145694.2145717

Christopher Claus and Walter Stechele. 2010. AutoVision—Reconfigurable Hardware Acceleration for Video-Based Driver Assistance. In *Dynamically Reconfigurable Systems*, Marco Platzner, Jürgen Teich, and Norbert Wehn (Eds.). Springer Netherlands, 375–394. DOI:http://dx.doi.org/10.1007/978-90-481-3485-4_18

J. Cong and Bingjun Xiao. 2013. Optimization of interconnects between accelerators and shared memories in dark silicon. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*. 630–637. DOI:http://dx.doi.org/10.1109/ICCAD.2013.6691182

Jason Cong and Yi Zou. 2009. FPGA-Based Hardware Acceleration of Lithographic Aerial Image Simulation. *ACM Trans. Reconfigurable Technol. Syst.* 2, 3, Article 17 (Sept. 2009), 29 pages. DOI:http://dx.doi.org/10.1145/1575774.1575776

Convey Computer. 2012. Convey Reference Manual. (2012).

John Curreri, Greg Stitt, and Alan George. 2012. Communication visualization for bottleneck detection of high-level synthesis applications. In *FPGA (FPGA '12)*. 33–36. DOI:http://dx.doi.org/10.1145/2145694.2145701

William J. Dally and Brian Towles. 2007. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers.

R. Das, S. Eachempati, A.K. Mishra, V. Narayanan, and C.R. Das. 2009. Design and evaluation of a hierarchical on-chip interconnect for next-generation CMPs. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*. 175–186. DOI:http://dx.doi.org/10.1109/HPCA.2009.4798252

B.C. Davis, P.T. Fletcher, E. Bullitt, and S. Joshi. 2007. Population Shape Regression From Random Design Data. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*. 1–7. DOI:http://dx.doi.org/10.1109/ICCV.2007.4408977

J. Diamond, M. Burtscher, J.D. McCalpin, Byoung-Do Kim, S.W. Keckler, and J.C. Browne. 2011. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. 32–43. DOI:http://dx.doi.org/10.1109/ISPASS.2011.5762713

C. Dick. 1996. Computing the Discrete Fourier Transform on FPGA Based Systolic Arrays. In *Field-Programmable Gate Arrays, 1996. FPGA '96. Proceedings of the 1996 ACM Fourth International Symposium on*. 129–135. DOI:http://dx.doi.org/10.1109/FPGA.1996.242440

B. Donchev, G. Kuzmanov, and G.N. Gaydadjiev. 2006. External Memory Controller for Virtex II Pro. In *System-on-Chip, 2006. International Symposium on*. 1–4. DOI:http://dx.doi.org/10.1109/ISSOC.2006.322009

Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. 2002. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Michel Dubois, Murali Annavaram, and Per Stenström. 2014. *Parallel Computer Organization and Design.* Cambridge University Press.

Hesham El-Rewini and Mostafa Abd-El-Barr. 2005. *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience.

Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. *SIGARCH Comput. Archit. News* 39, 3 (June 2011), 365–376. DOI:http://dx.doi.org/10.1145/2024723.2000108

University South Florida. 1999. Canny Edge Detector. (1999).

Fayez Gebali. 2011. *Interconnection Networks.* John Wiley & Sons, Inc., 83–103. DOI:http://dx.doi.org/10.1002/9780470932025.ch5

H. Giefers and M. Platzner. 2010. A Triple Hybrid Interconnect for Many-Cores: Reconfigurable Mesh, NoC and Barrier. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on.* 223–228. DOI:http://dx.doi.org/10.1109/FPL.2010.52

D. Goringer, M. Hubner, M. Benz, and J. Becker. 2010. A Design Methodology for Application Partitioning and Architecture Development of Reconfigurable Multiprocessor Systems-on-Chip. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on.* 259–262. DOI:http://dx.doi.org/10.1109/FCCM.2010.47

Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.* 17, 6 (June 1982), 120–126. DOI:http://dx.doi.org/10.1145/872726.806987

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. 2002. *Introduction to Parallel Computing* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Arnaud Grasset, Philippe Millet, Philippe Bonnot, Sami Yehia, Wolfram Putzke-Roeming, Fabio Campi, Alberto Rosti, Michael Huebner, NikolaosS. Voros, Davide Rossi, Henning Sahlbach, and Rolf Ernst. 2011. The MORPHEUS Heterogeneous Dynamically Reconfigurable Platform. *International Journal of Parallel Programming* 39, 3 (2011), 328–356. DOI:http://dx.doi.org/10.1007/s10766-010-0160-3

B. Grot, J. Hestness, S.W. Keckler, and O. Mutlu. 2009. Express Cube Topologies for on-Chip Interconnects. In *High Performance Computer Architecture, 2009. HPCA 2009.*

IEEE 15th International Symposium on. 163–174. DOI:http://dx.doi.org/10.1109/HPCA.2009.4798251

P. Guerrier and A. Greiner. 2000. A generic architecture for on-chip packet-switched interconnections. In *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*. 250–256. DOI:http://dx.doi.org/10.1109/DATE.2000.840047

John L. Gustafson. 1988. Reevaluating Amdahl's Law. *Commun. ACM* 31, 5 (May 1988), 532–533. DOI:http://dx.doi.org/10.1145/42411.42415

Linh K. Ha, Jens Kruger, Joao L. D. Comba, Claudio T. Silva, and Sarang Joshi. 2012. ISP: An Optimal Out-of-Core Image-Set Processing Streaming Architecture for Parallel Heterogeneous Systems. *IEEE Transactions on Visualization and Computer Graphics* 18, 6 (June 2012), 838–851. DOI:http://dx.doi.org/10.1109/TVCG.2012.32

Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. 2009. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing* 17 (2009), 242–254. DOI:http://dx.doi.org/10.2197/ipsjjip.17.242

H. Heideman, K. Collins, G. Vacek, and J. Bot. 2012. Eighteen- fold Performance Increase for Short-Read Sequence Mapping on. Genome of the Netherlands Data using Hybrid-Core Architecture. In *Netherlands Bioinformatics Conference (NBIC2012)*.

Timothy Heil, Anil Krishna, Nicholas Lindberg, Farnaz Toussi, and Steven Vanderwiel. 2014. Architecture and Performance of the Hardware Accelerators in IBM's PowerEN Processor. *ACM Trans. Parallel Comput.* 1, 1, Article 5 (May 2014), 26 pages. DOI:http://dx.doi.org/10.1145/2588888

J. Heisswolf, R. Konig, and J. Becker. 2012. 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*. 625–632. DOI:http://dx.doi.org/10.1109/ISPA.2012.93

D.L. Hung, Heng-Da Cheng, and S. Sengkhamyong. 1999. Design of a hardware accelerator for real-time moment computation: a wavefront away approach. *Industrial Electronics, IEEE Transactions on* 46, 1 (Feb 1999), 207–218. DOI:http://dx.doi.org/10.1109/41.744413

J.Y. Hur. 2011. *Customizing and Hardwiring On-Chip Interconnects in FPGAs*. Ph.D. Dissertation. Delft University of Technology, Delft, Netherlands. http://repository.tudelft.nl/view/ir/uuid%3A894e4234-ed2b-411f-be90-550766f97cd5/

JaeYoung Hur, Todor Stefanov, Stephan Wong, and Stamatis Vassiliadis. 2007. Systematic Customization of On-Chip Crossbar Interconnects. In *Reconfigurable Computing: Architectures, Tools and Applications*, PedroC. Diniz, Eduardo Marques, Koen Bertels, MarcioMerino Fernandes, and JoãoM.P. Cardoso (Eds.). Lecture Notes in Computer Science, Vol. 4419. Springer Berlin Heidelberg, 61–72. DOI:http://dx.doi.org/10.1007/978-3-540-71431-6_6

J. Y. Hur, T. Stefanov, S. Wong, and K. Goossens. 2012. Customisation of on-chip network interconnects and experiments in field-programmable gate arrays. *IET Computers Digital Techniques* 6, 1 (2012), 59–68. DOI:http://dx.doi.org/10.1049/iet-cdt.2010.0105

IBM. 1999. CoreConect Bus Architecture. (1999).

IBM. 2009. Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor. (2009). https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/7A77CCDF14FE70D5852575CA0074E8ED

Intel. 2010. Intel® Atom™ Processor E6x5C Series-based Platform for Embedded Computing. (2010). http://download.intel.com/embedded/processors/prodbrief/324535.pdf

S. Ishikawa, A. Tanaka, and T. Miyazaki. 2012. Hardware Accelerator for BLAST. In *Embedded Multicore Socs (MCSoC), 2012 IEEE 6th International Symposium on*. 16–22. DOI:http://dx.doi.org/10.1109/MCSoC.2012.22

A. Ismail and L. Shannon. 2011. FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. 170–177. DOI:http://dx.doi.org/10.1109/FCCM.2011.48

C. Jackson and S.J. Hollis. 2010. Skip-links: A dynamically reconfiguring topology for energy-efficient NoCs. In *System on Chip (SoC), 2010 International Symposium on*. 49–54. DOI:http://dx.doi.org/10.1109/ISSOC.2010.5625537

Natalie Enright Jerger and Li-Shiuan Peh. 2009. *On-Chip Networks* (1st ed.). Morgan and Claypool Publishers.

Yuho Jin, Eun Jung Kim, and T.M. Pinkston. 2012. Communication-Aware Globally-Coordinated On-Chip Networks. *Parallel and Distributed Systems, IEEE Transactions on* 23, 2 (Feb 2012), 242–254. DOI:http://dx.doi.org/10.1109/TPDS.2011.164

Tim Johnson and Umesh Nawathe. 2007. An 8-core, 64-thread, 64-bit Power Efficient Sparc Soc (Niagara2). In *Proceedings of the 2007 International Symposium on Physical Design (ISPD '07)*. ACM, New York, NY, USA, 2–2. DOI:http://dx.doi.org/10.1145/1231996.1232000

S. Jovanovic, C. Tanougast, S. Weber, and C. Bobda. 2007. CuNoC: A Scalable Dynamic NoC for Dynamically Reconfigurable FPGAs. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*. 753–756. DOI:http://dx.doi.org/10.1109/FPL.2007.4380761

Stamatis G. Kavadias, Manolis G.H. Katevenis, Michail Zampetakis, and Dimitrios S. Nikolopoulos. 2010. On-chip Communication and Synchronization Mechanisms with Cache-integrated Network Interfaces. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF '10)*. ACM, New York, NY, USA, 217–226. DOI:http://dx.doi.org/10.1145/1787275.1787328

J. Kim, J. Balfour, and W.J. Dally. 2007. Flattened Butterfly Topology for On-Chip Networks. *Computer Architecture Letters* 6, 2 (Feb 2007), 37–40. DOI:http://dx.doi.org/10.1109/L-CA.2007.10

Woo Joo Kim and Sun Young Hwang. 2008. Design of an Area-Efficient and Low-Power NoC Architecture Using a Hybrid Network Topology. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* E91-A, 11 (Nov. 2008), 3297–3303. DOI:http://dx.doi.org/10.1093/ietfec/e91-a.11.3297

Tim Kogel, Rainer Leupers, and Heinrich Meyr. 2006. Classification of platform elements. In *Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms*. Springer Netherlands, 15–32. DOI:http://dx.doi.org/10.1007/1-4020-4826-2_3

Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. 2005. Heterogeneous Chip Multiprocessors. *Computer* 38, 11 (Nov. 2005), 32–38. DOI:http://dx.doi.org/10.1109/MC.2005.379

Steve Leibson. 2014. What's the Right Road to ASIC-Class Status for FPGAs? (May 2014). http://www.xilinx.com/publications/archives/xcell/Xcell86.pdf

Liu Ling, Neal Oliver, Chitlur Bhushan, Wang Qigang, Alvin Chen, Shen Wenbo, Yu Zhihong, Arthur Sheiman, Ian McCallum, Joseph Grecco, Henry Mitchel, Liu Dong, and Prabhat Gupta. 2009. High-performance, Energy-efficient Platforms Using In-socket FPGA Accelerators. In *FPGA (FPGA '09)*. 261–264. DOI:http://dx.doi.org/10.1145/1508128.1508172

Haisheng Liu, Smail Niar, Yassin El-Hillali, and Atika Rivenq. 2011. Embedded Architecture with Hardware Accelerator for Target Recognition in Driver Assistance System. *SIGARCH Comput. Archit. News* 39, 4 (Dec. 2011), 56–59. DOI:http://dx.doi.org/10.1145/2082156.2082170

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* 40, 6 (June 2005), 190–200. DOI:http://dx.doi.org/10.1145/1064978.1065034

Roman Lysecky and Frank Vahid. 2009. Design and Implementation of a MicroBlaze-based Warp Processor. *ACM Trans. Embed. Comput. Syst.* 8, 3, Article 22 (April 2009), 22 pages. DOI:http://dx.doi.org/10.1145/1509288.1509294

R. Manevich, I. Walter, I. Cidon, and A. Kolodny. 2009. Best of both worlds: A bus enhanced NoC (BENoC). In *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*. 173–182. DOI:http://dx.doi.org/10.1109/NOCS.2009.5071465

Débora Matos, Caroline Concatto, and Luigi Carro. 2013. Reconfigurable Intercommunication Infrastructure: NoCs. In *Adaptable Embedded Systems*, Antonio Carlos Schneider Beck, Carlos Arthur Lang Lisbôa, and Luigi Carro (Eds.). Springer New York, 119–161. DOI:http://dx.doi.org/10.1007/978-1-4614-1746-0_5

M. Modarressi, A. Tavakkol, and H. Sarbazi-Azad. 2010. Virtual Point-to-Point Connections for NoCs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 29, 6 (June 2010), 855–868. DOI:http://dx.doi.org/10.1109/TCAD.2010.2048402

S. Murali, L. Benini, and G. De Micheli. 2007. An Application-Specific Design Methodology for On-Chip Crossbar Generation. *Computer-Aided Design of Integrated Circuits and Systems* 26, 7 (2007), 1283–1296. DOI:http://dx.doi.org/10.1109/TCAD.2006.888284

Srinivasan Murali, Paolo Meloni, Federico Angiolini, David Atienza, Salvatore Carta, Luca Benini, Giovanni De Micheli, and Luigi Raffo. 2006. Designing Application-specific Networks on Chips with Floorplan Information. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design (ICCAD '06)*. ACM, New York, NY, USA, 355–362. DOI:http://dx.doi.org/10.1145/1233501.1233573

R. Nane, V. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. 2012. DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. 619–622. DOI:http://dx.doi.org/10.1109/FPL.2012.6339221

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. DOI:http://dx.doi.org/10.1145/1273442.1250746

S. Nilakantan, S. Battle, and M. Hempstead. 2013. Metrics for Early-Stage Modeling of Many-Accelerator Architectures. *Computer Architecture Letters* 12, 1 (January 2013), 25–28. DOI:http://dx.doi.org/10.1109/L-CA.2012.9

Juan Manuel Orduña, Federico Silla, and José Duato. 2004. On the Development of a Communication-aware Task Mapping Technique. *J. Syst. Archit.* 50, 4 (March 2004), 207–220. DOI:http://dx.doi.org/10.1016/j.sysarc.2003.09.002

S.A. Ostadzadeh. 2012. *Quantitative Application Data Flow Characterization for Heterogeneous Multicore Architectures*. Ph.D. Dissertation. Delft University of Technology, Delft, Netherlands. DOI:http://dx.doi.org/doi:10.4233/uuid:5acbd4cd-ab5f-486a-8268-61cd7416550d

S.Arash Ostadzadeh, Roel Meeuws, Imran Ashraf, Carlo Galuzzi, and Koen Bertels. 2012. The Q2 Profiling Framework: Driving Application Mapping for Heterogeneous Reconfigurable Platforms. In *Reconfigurable Computing: Architectures, Tools and Applications*, OliverC.S. Choy, RayC.C. Cheung, Peter Athanas, and Kentaro Sano (Eds.). Lecture Notes in Computer Science, Vol. 7199. Springer Berlin Heidelberg, 76–88. DOI:http://dx.doi.org/10.1007/978-3-642-28365-9_7

V. Papaefstathiou, D. Pnevmatikatos, M. Marazakis, G. Kalokairinos, A. Ioannou, M. Papamichael, S. Kavadias, G. Mihelogiannakis, and M. Katevenis. 2007. Prototyping Efficient Interprocessor Communication Mechanisms. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*. 26–33. DOI:http://dx.doi.org/10.1109/ICSAMOS.2007.4285730

Sudeep Pasricha and Nikil Dutt. 2008. Chapter 2 - Basic Concepts of Bus-Based Communication Architectures. In *On-Chip Communication Architectures*, Sudeep Pasricha and Nikil Dutt (Eds.). Morgan Kaufmann, 17 – 41. DOI:http://dx.doi.org/10.1016/B978-0-12-373892-9.00002-5

Oliver Pell and Oskar Mencer. 2011. Surviving the End of Frequency Scaling with Reconfigurable Dataflow Computing. *SIGARCH Comput. Archit. News* 39, 4 (Dec. 2011), 60–65. DOI:http://dx.doi.org/10.1145/2082156.2082172

Dac Pham, Jim Holt, and Sanjay Deshpande. 2011. Embedded Multicore Systems: Design Challenges and Opportunities. In *Multiprocessor System-on-Chip*, Michael Hübner and Jürgen Becker (Eds.). Springer New York, 197–222. DOI:http://dx.doi.org/10.1007/978-1-4419-6460-1_9

C. Pilato, A. Cazzaniga, G. Durelli, A. Otero, D. Sciuto, and M.D. Santambrogio. 2012. On the automatic integration of hardware accelerators into FPGA-based embedded systems. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. 607–610. DOI:http://dx.doi.org/10.1109/FPL.2012.6339218

Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *FPGA (FPGA '13)*. 29–38. DOI:http://dx.doi.org/10.1145/2435264.2435273

Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ISCA*.

T.D. Richardson, C. Nicopoulos, Dongkook Park, V. Narayanan, Yuan Xie, C. Das, and V. Degalahal. 2006. A hybrid SoC interconnect with dynamic TDMA-based transaction-less buses and on-chip networks. In *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*. 8 pp.–. DOI:http://dx.doi.org/10.1109/VLSID.2006.10

A. Roca, J. Flich, and G. Dimitrakopoulos. 2012. DESA: Distributed Elastic Switch Architecture for efficient networks-on-FPGAS. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. 394–399. DOI:http://dx.doi.org/10.1109/FPL.2012.6339135

Carlos Alberto Rueda-Velasquez. 2007. *Geospatial Image Stream Processing: Models, Techniques, and Applications in Remote Sensing Change Detection*. Ph.D. Dissertation. Davis, CA, USA. AAI3303196.

MateusBeck Rutzig. 2013. Multicore Platforms: Processors, Communication and Memories. In *Adaptable Embedded Systems*, Antonio Carlos Schneider Beck, Carlos Arthur Lang Lisbôa, and Luigi Carro (Eds.). Springer New York, 243–277. DOI:http://dx.doi.org/10.1007/978-1-4614-1746-0_8

Daniel Sanchez, George Michelogiannakis, and Christos Kozyrakis. 2010. An Analysis of On-chip Interconnection Networks for Large-scale Chip Multiprocessors. *ACM Trans. Archit. Code Optim.* 7, 1, Article 4 (May 2010), 28 pages. DOI:http://dx.doi.org/10.1145/1756065.1736069

S. Sarkar, T. Majumder, A Kalyanaraman, and P.P. Pande. 2010. Hardware accelerators for biocomputing: A survey. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. 3789–3792. DOI:http://dx.doi.org/10.1109/ISCAS.2010.5537736

Tobias Schumacher, Christian Plessl, and Marco Platzner. 2012. IMORC: An infrastructure and architecture template for implementing high-performance reconfigurable FPGA accelerators. *Microprocessors and Microsystems* 36, 2 (2012), 110 – 126. DOI:http://dx.doi.org/10.1016/j.micpro.2011.04.002

Jeff Scott, Lea Hwang Lee, John Arends, and Bill Moyer. 1998. Designing the Low-Power M•CORE Architecture. In *ISCA Workshop*.

J. Shi and C. Tomasi. 1994. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*. 593–600. DOI:http://dx.doi.org/10.1109/CVPR.1994.323794

Amit Kumar Singh, Thambipillai Srikanthan, Akash Kumar, and Wu Jigang. 2010. Communication-aware Heuristics for Run-time Task Mapping on NoC-based MPSoC Platforms. *J. Syst. Archit.* 56, 7 (July 2010), 242–255. DOI:http://dx.doi.org/10.1016/j.sysarc.2010.04.007

Stephen M. Smith. 1992. SUSAN Low Level Image Processing. (1992).

Polar SSL. 2012. PolarSSL library. (2012). http://polarssl.org/

Jos Stam. 2003. Real-time Fluid Dynamics for Games. In *Game Developer Conference*.

M.B. Stensgaard and J. Sparso. 2008. ReNoC: A Network-on-Chip Architecture with Reconfigurable Topology. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*. 55–64. DOI:http://dx.doi.org/10.1109/NOCS.2008.4492725

J. Stuecheli. August 2013. Next Generation POWER microprocessor. In *HotChips 2013*.

K. Swaminathan, S. Gopi, Rajkumar, G. Lakshminarayanan, and Seok-Bum Ko. 2014. A novel hybrid topology for Network on Chip. In *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on*. 1–6. DOI:http://dx.doi.org/10.1109/CCECE.2014.6901083

V. Todorov, D. Mueller-Gritschneder, H. Reinig, and U. Schlichtmann. 2014. Deterministic Synthesis of Hybrid Application-Specific Network-on-Chip Topologies. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 33, 10 (Oct 2014), 1503–1516. DOI:http://dx.doi.org/10.1109/TCAD.2014.2331556

Kun-Lin Tsai, Feipei Lai, Chien-Yu Pan, Di-Sheng Xiao, Hsiang-Jen Tan, and Hung-Chang Lee. 2010. Design of low latency on-chip communication based on hybrid NoC architecture. In *NEWCAS Conference (NEWCAS), 2010 8th IEEE International*. 257–260. DOI:http://dx.doi.org/10.1109/NEWCAS.2010.5603934

S. Vassiliadis and I. Sourdis. 2006. FLUX Networks: Interconnects on Demand. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2006. IC-SAMOS 2006. International Conference on*. 160–167. DOI:http://dx.doi.org/10.1109/ICSAMOS.2006.300823

S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E.M. Panainte. 2004. The MOLEN polymorphic processor. *Computers, IEEE Transactions on* 53, 11 (Nov 2004), 1363–1375. DOI:http://dx.doi.org/10.1109/TC.2004.104

Nikolaos S. Voros, Michael Hübner, Jürgen Becker, Matthias Kühnle, Florian Thomaitiv, Arnaud Grasset, Paul Brelet, Philippe Bonnot, Fabio Campi, Eberhard Schüler, Henning Sahlbach, Sean Whitty, Rolf Ernst, Enrico Billich, Claudia Tischendorf, Ulrich Heinkel, Frank Ieromnimon, Dimitrios Kritharidis, Axel Schneider, Joachim Knaeblein, and Wolfram Putzke-Röming. 2013. MORPHEUS: A Heterogeneous Dynamically Reconfigurable Platform for Designing Highly Complex Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 12, 3, Article 70 (April 2013), 33 pages. DOI: http://dx.doi.org/10.1145/2442116.2442120

Chifeng Wang, Wen-Hsiang Hu, Seung Eun Lee, and Nader Bagherzadeh. 2011. Area and power-efficient innovative congestion-aware Network-on-Chip architecture. *Journal of Systems Architecture* 57, 1 (2011), 24 – 38. DOI:http://dx.doi.org/10.1016/j.sysarc.2010.10.009 Special Issue On-Chip Parallel And Network-Based Systems.

Ruediger Willenberg and Paul Chow. 2013. A remote memory access infrastructure for global address space programming models in FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '13)*. 211–220. DOI:http://dx.doi.org/10.1145/2435264.2435301

Xilinx. 2009. ML510 Reference Design. (2009).

Xilinx. 2014. Vivado High-Level Synthesis. (2014).

Jieming Yin, Pingqiang Zhou, Sachin S. Sapatnekar, and Antonia Zhai. 2014. Energy-Efficient Time-Division Multiplexed Hybrid-Switched NoC for Heterogeneous Multicore Systems. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE Computer Society, Washington, DC, USA, 293–303. DOI:http://dx.doi.org/10.1109/IPDPS.2014.40

Heng Yu, Yajun Ha, and B. Veeravalli. 2010. Communication-aware application mapping and scheduling for NoC-based MPSoCs. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. 3232–3235. DOI:http://dx.doi.org/10.1109/ISCAS.2010.5537920

Zhongda Yuan, Yuchun Ma, and Jinian Bian. 2012. SMPP: Generic SAT Solver over Reconfigurable Hardware Accelerator. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*. 443–448. DOI: http://dx.doi.org/10.1109/IPDPSW.2012.57

Payman Zarkesh-Ha, George B.P. Bezerra, Stephanie Forrest, and Melanie Moses. 2010. Hybrid Network on Chip (HNoC): Local Buses with a Global Mesh Architecture. In *Proceedings of the 12th ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP '10)*. ACM, New York, NY, USA, 9–14. DOI:http://dx.doi.org/10.1145/1811100.1811104

Hui Zhao, Ohyoung Jang, Wei Ding, Yuanrui Zhang, Mahmut Kandemir, and Mary Jane Irwin. 2012. A Hybrid NoC Design for Cache Coherence Optimization for Chip Multiprocessors. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 834–842. DOI:http://dx.doi.org/10.1145/2228360.2228511

SAMENVATTING

Heterogene multicore-systemen worden steeds belangrijker naar mate de behoefte aan rekenkracht groeit, vooral nu we het big data tijdperk betreden. Hardware accelerator systemen zijn een van de belangrijkste trends binnen heterogeneous multicore en bieden applicatie specifieke hardware schakelingen, waardoor deze schakelingen energie-efficiënter zijn en hogere prestaties bieden dan general purpose processors, terwijl ze toch een grote mate van flexibiliteit bieden. Echter, de systeemprestaties zijn niet schaalbaar wanneer processing cores worden toegevoegd, wat is te wijten aan de communicatie overhead die sterk toeneemt met het toenemend aantal cores. Alhoewel datacommunicatie een primair verwacht knelpunt is voor de systeem prestaties, is het ontwerp van interconnects voor datacommunicatie tussen de hardware accelerator kernels nog niet goed behandeld voor hardwareversneller systemen. Meestal wordt een eenvoudige bus of gedeeld geheugen gebruikt voor de datacommunicatie tussen de accelerator kernels. In dit proefschrift behandelen we het probleem van het interconnect ontwerp voor heterogene hardwareversneller systemen.

Het is duidelijk dat er afhankelijkheden zijn tussen berekeningen, aangezien gegevens geproduceerd door de ene kernel nodig kunnen zijn voor een andere. Datacommunicatie patronen kunnen specifiek zijn voor elke toepassing en kunnen leiden tot verschillende interconnects. In dit proefschrift gebruiken we gedetailleerde datacommunicatie profilering om een geoptimaliseerde hybride interconnect te ontwerpen dat de meest geschikte ondersteuning biedt voor het communicatie patroon binnen een applicatie, terwijl we het gebruik van hardware resources voor de interconnect proberen te minimalizeren. Ten eerste stellen we een heuristische benadering voor waarbij rekening wordt gehouden met de datacommunicatie profilering van de toepassing om een hardware accelerator systeem te ontwerpen met een aangepaste interconnect. Een aantal oplossingen worden beschouwd, waaronder op crossbar-gebaseerd gedeeld lokaal geheugen, Direct Memory Access (DMA)-ondersteund parallel processing, lokale buffers, en hardware duplicatie. Deze aanpak is vooral nuttig voor embedded systemen waar de hardware resources beperkt zijn. Ten tweede stellen we een

geautomatiseerde hybride interconnect ontwerp voor om met behulp van data-communicatie profilering een geoptimaliseerde interconnect te definiëren voor accelerator kernels van een generieke hardware accelerator systeem. De hybride interconnect bestaat uit een netwerk-op-chip (NOC), gedeelde lokaal geheugen, of beide. Om hardware resource gebruik voor de hybride interconnect te minimaliseren, stellen we ook een adaptief mapping algoritme voor om computing kernels en hun lokale geheugens te verbinden aan de voorgestelde hybride interconnect. Ten derde stellen we een hardwareversneller architectuur voor ter ondersteuning van streaming beeldverwerking. Voor alle gepresenteerde benaderingen implementeren we de aanpak met behulp van een aantal benchmarks op relevante herconfigureerbare platforms om hun effectiviteit te tonen. De experimentele resultaten laten zien dat onze benadering niet alleen de systeemprestaties kunnen verbeteren, maar ook kunnen leiden tot een vermindering van het totale energieverbruik ten opzichte van de baseline systemen.

LỜI CẢM ƠN

Thật không dễ dàng để viết nên những lời cảm ơn trong quyển luận văn này; nhưng cũng thật là thú vị khi dành thời gian nhìn lại một cách cẩn thận những gì đã qua trong suốt bốn năm qua, bắt đầu từ năm 2011. Đầu tiên, xin gửi lời cảm ơn đến Cục Đào tạo với Nước ngoài vì đã hỗ trợ tài chính cho tôi. Không có sự hỗ trợ này, tôi đã không thể đến Hà Lan để học tập.

Tôi muốn dành lời cảm ơn chân thành nhất cho thầy tôi, GS. TS. Koen Bertels, người đã phải đưa ra một quyết định khó khăn, nhưng cũng là một quyết định thành công, khi chấp nhận tôi làm nghiên cứu sinh với ông vào năm 2011. Ở thời điểm đó, tôi nói tiếng Anh chưa được thật tốt, tuy nhiên ông vẫn cố gắng hết sức để hiểu trong cuộc phỏng vấn qua Skype giữa chúng tôi. Trong suốt thời gian làm nghiên cứu sinh, ông đã giới thiệu cho tôi nhiều ý tưởng rất hay và để tôi tự do trong nghiên cứu. Không có ông tôi đã không có cơ hội để viết ra quyển luận văn này. Tôi cũng muốn cảm ơn chân thành thầy đồng hướng dẫn, TS. Zaid Al-Ars, nhưng ông luôn gọi và xem tôi như là bạn của ông. Ông đã hướng dẫn cho tôi rất nhiều từ việc nghiên cứu đến viết những bài báo khoa học. Tôi sẽ không bao giờ quên những lúc ông dành hàng giờ liền để sửa bài cho tôi. Không có ông tôi không thể xuất bản bất kỳ bài báo nào, và tất nhiên không thể có quyển luận văn này. Bên cạnh hai thầy, tôi xin cảm ơn Veronique - trung tâm Valorisation, Lidwina - thư ký phòng thí nghiệm kỹ thuật máy tính, Eef và Erik - những người quản trị hệ thống ở phòng thí nghiệm bởi những hỗ trợ cho tôi trong suốt thời gian qua. Tôi cũng cảm ơn Razvan về chương trình dịch DWARV và Vlad về hệ thống Molen mà dựa trên đó tôi đã thực hiện những thí nghiệm của tôi.

Cuộc sống không phải chỉ có làm việc và nghiên cứu. Không có những khoảng thời gian thư giản thì chúng ta sẽ không có năng lượng để làm việc cũng như chẳng có những ý tưởng để thành công. Do đó, tôi muốn cảm ơn các anh em trong nhóm ANCB về những khoảng thời gian rất thoải mái mà chúng ta đã cùng tạo nên. Những bữa tiệc và những khoảng thời gian

thư giãn đó đã giúp tôi nạp đầy năng lượng sau những ngày làm việc căng thẳng. Tôi không thể nào nêu hết ra đây những người đã giúp đỡ tôi trong suốt bốn năm qua vì sẽ mất hàng trăm trang nếu tôi làm như thế; nhưng tôi cũng chắc chắn rằng tôi không bao giờ có thể quên. Xin cho phép tôi giữ những điều tốt đẹp đó trong đầu mình.

Một lời cảm ơn sâu sắc xin gửi tới gia đình tôi và gia đình bên vợ tôi, đặc biệt là ba và mẹ vợ tôi vì đã chăm sóc con tôi trong thời gian tôi xa nhà. Không có ông bà ngoại tôi đã không thể bình yên để làm việc.

Cuối cùng nhưng là lời quan trọng nhất, ngàn lời cảm ơn vợ và con. Hai mẹ con là nguồn động viên và giúp ba thêm sức mạnh. Không có tình yêu thương từ hai mẹ con thì ba không hoàn thành công việc đúng thời hạn. Gia đình chúng ta sắp đoàn tụ trong một vài tháng tới sau khoảng thời gian dài phải sống bên nhau bằng các cuộc điện thoại, thư điện tử, mạng xã hội, và qua những chuyến đi.

Phạm Quốc Cường
Delft, Mùa xuân 2015

LIST OF PUBLICATIONS

9. ***C. Pham-Quoc**, I. Ashraf, Z. Al-Ars, K.L.M. Bertels,* **Data Communication Driven Hybrid Interconnect Design for Heterogeneous Hardware Accelerator Systems**, Submitted to ACM Transactions on Reconfigurable Technology and Systems,
DOI:.

8. *R. Nane, V.M. Sima, **C. Pham-Quoc**, F Goncalves, K.L.M. Bertels*, **High-Level Synthesis in the Delft Workbench Hardware/Software Co-design Tool-Chain**, 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC 2014), 26-28 August 2014, Milan, Italy,
DOI:10.1109/EUC.2014.28.

7. ***C. Pham-Quoc**, Z. Al-Ars, K.L.M. Bertels*, **Automated Hybrid Interconnect Design for FPGA Accelerators Using Data Communication Profiling**, 28th International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2014), 19-23 May 2014, Phoenix, USA,
DOI:10.1109/IPDPSW.2014.21.

6. ***C. Pham-Quoc**, Z. Al-Ars, K.L.M. Bertels*, **Heterogeneous Hardware Accelerator Architecture for Streaming Image Processing**, International Conference on Advanced Technologies for Communications (ATC 2013), 16-18 October 2013, Ho Chi Minh City, Vietnam,
DOI:10.1109/ATC.2013.6698140.

5. ***C. Pham-Quoc**, Z. Al-Ars, K.L.M. Bertels*, **Heterogeneous Hardware Accelerators Interconnect: An Overview**, NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2013), 25-27 June 2013, Torino, Italy,
DOI:10.1109/AHS.2013.6604245.

4. ***C. Pham-Quoc**, J. Heisswolf, S. Wenner, Z. Al-Ars, J.A. Becker, K.L.M. Bertels,* **Hybrid Interconnect Design for Heterogeneous Hardware Accelerators**, Design, Automation & Test in Europe Conference & Exhibition (DATE

2013), 18-22 March 2013, Grenoble, France,
DOI:10.7873/DATE.2013.178.

3. ***C. Pham-Quoc**, Z. Al-Ars, K.L.M. Bertels*, **Heterogeneous Hardware Accelerators Interconnect: An Overview**, 7th HiPEAC Workshop on Reconfigurable Computing (WRC 2013), 21 January 2013, Berlin, Germany.

2. ***C. Pham-Quoc**, Z. Al-Ars, K.L.M. Bertels*, **Rule-Based Data Communication Optimization Using Quantitative Communication Profiling**, International Conference on Field-Programmable Technology (FPT 2012), 10-12 December 2012, Seoul, Korea,
DOI:10.1109/FPT.2012.6412119.

1. ***C. Pham-Quoc**, Z. Al-Ars, K.L.M. Bertels*, **A Heuristic-based Communication-aware Hardware Optimization Approach in Heterogeneous Multicore Systems**, International Conference on ReConFigurable Computing and FPGAs (ReConFig 2012), 5-7 December 2012, Cancun, Mexico,
DOI:10.1109/ReConFig.2012.6416720.

CURRICULUM VITÆ

Cuong Pham-Quoc (**Phạm Quốc Cường**, in Vietnamese) was born on 28 March 1985 in Tien Giang, Vietnam. He received his B.Sc. in Computer Science and Engineering and his M.Eng. in Computer Science both from Ho Chi Minh City University of Technology (HCMUT), Vietnam National Unviersity, Vietnam in 2007 and 2009, respectively. In May 2007, he had started working as an assistant lecturer in the Faculty of Computer Science and Engineering, HCMUT before became a lecturer at the same faculty in 2008. In May 2011, he received a scholarship from Vietnam International Education Development (VIED) and joined the Computer Engineering Lab, Delft University of Technology as a PhD student with the supervision of Dr. Zaid Al-Ars and Prof. Dr. Koen Bertels. Currently, his research interest include: reconfigurable computing, hybrid interconnect, multi/many cores architecture, hardware/software co-design.