RECONFIGURABLE NETWORK-ON-CHIP (NoC) ARCHITECTURES FOR EMBEDDED SYSTEMS

by

Salih Bayar

B.S., Electronics and Communication Engineering, Yıldız Technical University, 2003, İstanbul, Turkey

M.S., Systems Engineering, Electronics and Information Technology, University of Karlsruhe(TH), 2007, Karlsruhe, Germany

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering
Boğaziçi University
2015

RECONFIGURABLE NETWORK-ON-CHIP (NoC) ARCHITECTURES FOR EMBEDDED SYSTEMS

APPROVED BY:

Assoc. Prof. Arda Yurdakul (Thesis Supervisor)	.....
Prof. Oğuz Tosun	.....
Assoc. Prof. Sıddıka Berna Örs Yalçın	.....
Assoc. Prof. Alper Şen	.....
Assist. Prof. Faik Başkaya	.....

DATE OF APPROVAL: 16.12.2014

To the memory of my beloved Father...

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Assoc. Prof. Arda Yurdakul. Without her guidance and support, it would be impossible to come to the end of this thesis.

I am also very grateful to Prof. Oğuz Tosun and Assoc. Prof. Sıddıka Berna Örs Yalçın because of their valuable support and recommendations during my thesis. I would also like to thank Assoc. Prof. Alper Şen and Assist. Prof. Faik Başkaya for their participation in my thesis committee and for their helpful comments to improve this thesis.

I would like to express my very great appreciations to Dağhan Dinç for his valuable help in the development of Systematic Resampling Algorithm. Advices given by Coşkun Çelik and Assoc. Prof. Süleyman Tosun have been great help in modifying NIRGAM NoC Simulator. I would like to thank Ömer Çoğal for supplying numeric test values for NoC, Prof. Smail Niar and Assist. Prof. Betül Demiröz for their supports, suggestions and corrections in case studies.

This thesis work is supported by The Scientific and Technological Research Council of Turkey, TÜBİTAK (Project Nr.: 104E038), Boğaziçi University Scientific Research Projects (Project Nr.: 06M105, 09A103P, 5578) and State Planning Organization of Turkey, DPT under grant no 2007K120610.

ABSTRACT

RECONFIGURABLE NETWORK-ON-CHIP (NoC) ARCHITECTURES FOR EMBEDDED SYSTEMS

Communication architectures such as Point-to-Point (P2P) and shared bus are poorly scalable as the number of cores or the communication volume increase. Network-on-Chip (NoC) has been proposed to reduce power consumption and has been widely adopted by the System-on-Chip (SoC) community. Yet, NoCs occupy more area and consume more power as the size of network increases. In this thesis, we propose a novel dynamic reconfigurable P2P (DRP2P) communication architecture for reconfigurable embedded systems, which is an alternative to the conventional NoC architectures. In DRP2P, interconnects are reconfigured on-the-fly as new communication requests arrive at the system. In embedded applications running on the multi-core systems, the traffic flow is usually known. Hence, DRP2P is very suitable for embedded systems. DRP2P is inspired from both P2P interconnects and NoC architecture. If the traffic flow is known in advance, it works as fast as P2P while reconfiguration process is done at the time of computation. Thus, next communication scenario can be established before communication starts. Since the reconfigurable wiring area in DRP2P is proportional to the network size, it is as scalable as NoC. In order to achieve reconfiguration efficiently, we developed three different dedicated self reconfiguration engines. The latest version of these engines is exploited in DRP2P architecture. DRP2P gives better results than conventional NoCs if the physical placement of cores on the embedded system is done properly by utilizing mapping and routing algorithms. Hence, fast and heuristic mapping and routing algorithms are also designed in the scope of this thesis. Experimental evaluations have shown that DRP2P outperforms conventional NoCs even in the worst case scenario as the amount of data in on-chip communication increases.

ÖZET

GÖMÜLÜ SİSTEMLER İÇİN YENİDEN BETİMLENEBİLİR YONGA ÜSTÜ AĞ (YüA) MİMARİLERİ

Noktadan noktaya (NN) ya da paylaşımlı veri yolu gibi haberleşme mimarileri, çekirdek sayısı ve bu çekirdekler arasındaki iletişim hacmi arttıkça ölçeklenememektedir. Güç tüketimini azaltmak için, Yonga-üstü-Ağ (YüA) mimarileri öne sürülmüş olup, bu mimariler Yonga-üstü-Sistem topluluğu tarafından yaygın olarak kabul görmüştür. Ancak ağ boyutu arttıkça, YüA mimarileri daha fazla alan kaplamakta ve daha çok güç tüketmektedirler. Bu yüzden, bu tezde, geleneksel YüA mimarilerine alternatif olarak, Dinamik Yeniden betimlenebilir Noktadan Noktaya (DYNN) mimariler sunulmaktadır. DYNN'de sisteme yeni haberleşme istekleri geldiğinde, bağlantılar dinamik olarak yeniden betimlenir. Çok çekirdekli sistemler üzerinde koşan Gömülü Sistem (GS) uygulamalarında, trafik akışı genellikle önceden bilinmektedir. DYNN mimarisi hem NN hem de YüA mimarilerinden esinlenerek tasarlanmıştır. Eğer trafik akışı önceden bilinirse, yeniden betimleme (YB) işlemi hesaplama zamanında yapıldığından, DYNN, NN kadar hızlı çalışır. Böylece, bir sonraki haberleşme senaryosu, haberleşme başlamadan kurulabilir. DYNN'de YB alanı ağ boyutu ile doğru orantılı olduğundan dolayı, DYNN, geleneksel YüA gibi ölçeklenebilmektedir. Etkin bir YB için, tez kapsamında üç adet YB motoru tasarlanmıştır. Bu motorların en son sürümü DYNN'de kullanılmış olup, hedef sistem tarafından desteklenen en yüksek hızda çalışabilmektedir. Eğer GS üzerinde çekirdeklerin yerleşmesi etkin eşleme ve yönlendirme algoritmaları kullanılarak yapılırsa, DYNN, geleneksel YüA'lardan daha iyi sonuçlar vermektedir. Bu yüzden, tez kapsamında sezgisel eşleme ve yönlendirme algoritmaları tasarlanmıştır. Deneysel sonuçlara göre, DYNN'nin yonga üstü haberleşmede verinin arttığı en kötü durumda bile, geleneksel YüA'dan daha iyi çalıştığı gözlemlenmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

ADR_{p_i}	Address of p_i^c
C	Computational core set/ Number of columns in a regular 2-D mesh NoC architecture
c	Vertical index of a node in regular 2-D mesh NoC architecture
CC_{se}	Communication cost for a single edge
$dist_{i,j}$	Dijkstra's shortest path
$E_{bit}^{t_i,t_j}$	Average energy consumption of sending one bit of data from one node to another node
E_{comm}	Total communication energy
E_{LHbit}	Energy consumed on horizontal links
E_{LVbit}	Energy consumed on vertical links
$E_{L_{bit}}$	Energy consumed by switches between tiles
E_{Rbit}	Energy consumed by a router
$E_{S_{bit}}$	Energy consumed by switches between tiles
$f_{i,j}$	Traffic flow from node i to node j
G_x	Subset of I^S and it contains partial bitstreams p_i that have to be stored in BRAM
I^S	Optimal set with N compressed bitstreams
i	Threshold value
K_m	The set of all subsets with m members from I
L_{comm}	Total communication latency
$l_{i,j}$	Link between cores c_i and c_j
l_{ij}	Length of joint partial bitstream of i^{th} and j^{th} bitstreams
l_i	Length of i^{th} partial bitstream
$MDist$	Manhattan Distance
N	Number of communication scenarios/ Number of task nodes
n	Number of routers
n_{hops}	Number of hops
n_H	Number of horizontal links

n_i	i^{th} task node
n_V	Number of vertical links
P	Number of physical processors
p_{ij}	Joint bitstream of p_i and p_j
p_{ij}^c	Compressed version of joint partial bitstream p_{ij}
p_i	Partial bitstream that is generated by reconfiguration of a region
p_i^c	Compressed version of partial bitstream p_i
p_j	Partial bitstream that is generated by reconfiguration of the same region with p_i after a period of time
$P_{r,c}$	Physical location of a processor with row index r and column index c
r	Horizontal index of a node in regular 2-D mesh NoC architecture
R	Number of rows in a regular 2-D mesh NoC architecture
S_D	Data transfer size
T	Number of non-zero directed edges between nodes n_i and n_j
T_{clk}	Cycle time of the system clock
T_{comm}	Communication time
T_{comp}	Computation time
$t_{i,j}$	Traffic amount from node i to node j
t_i	Tile i
T_{reconf}	Reconfiguration time
W_c	Channel width between modules (in bits)

LIST OF ACRONYMS/ABBREVIATIONS

AI	Automotive Industry
APU	Accelerated Processing Unit
ASP2P	Application Specific Point to Point
BRAM	Block RAM
BB	Branch-and-Bound
BM	Bus Macro
CCLK	Configuration Clock
CLB	Configurable Logic Block
cPCAP	compressed Parallel Configuration Access Port
CPLD	Complex Programmable Logic Device
CR	Complete Reconfiguration/Compression Ratio/Clock Region
CS	Compressed Size
CTG	Core Traffic Graph
DCM	Digital Clock Manager
DMA	Direct Memory Access
DPSR	Dynamic Partial Self-Reconfiguration
DPT	State Planning Organization of Turkey
DRP	Dynamic Reconfiguration Port
DRP2P	Dynamically Reconfigurable Point to Point Interconnects
DSD	Dual Screen Display
DTCNN	Discrete Time Cellular Neural Network
DyNoC	Dynamic Network on Chip
E3S	Embedded Systems Synthesis Benchmarks Suite
EIB	Element Interconnect Bus
FSM	Finite State Machine
FPGA	Field-Programmable Gate Array
FPL	Field Programmable Logic
GMAP	Greedy Mapping Algorithm
GPIO	General Purpose Input Output

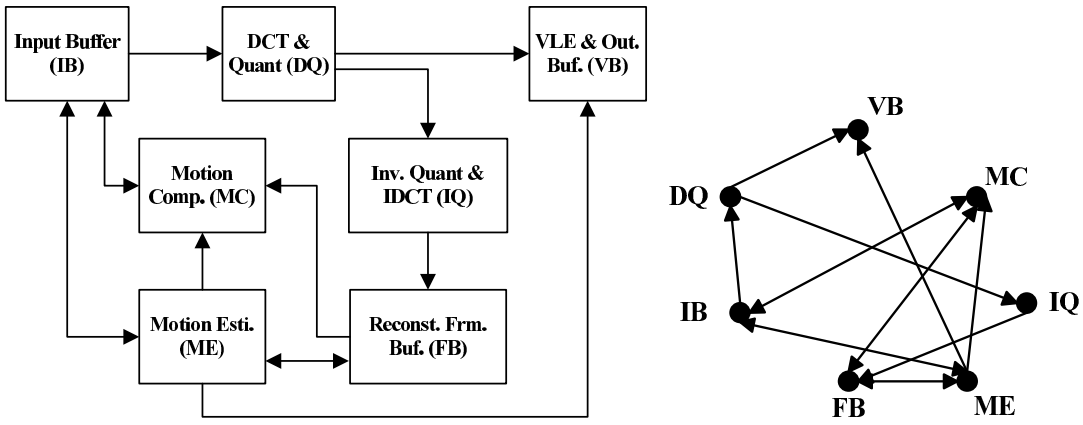
GPIPS	General Purpose Image Processing System
GPU	Graphic Processing Unit
HBM	Hard Bus Macro
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
IT	Iteration Count
JCAP	Virtual Internal Configuration Access Port
LB	Lower Bound
MDist	Manhattan Distance
MIC	Memory Interface Controller
MMS	Multimedia System
MMS25	Multimedia System with 25-Cores
MMS40	Multimedia System with 40-Cores
MPSoC	Multiprocessors System-on-Chip
MST	Master Burst
MT	Mersenne Twister
MTT	Multiple Target Tracking
MWAG	Multi-Window Application with Graphics
MWA	Multi-Window Application
MWD	Multi-Window Display
NCA	Node Configuration Architecture
NCD	Native Circuit Description
NCG	Node Configuration Graph
NoCem	Network on Chip emulator
NoC	Network on Chip
OP	Optimum Solution
OPB	On-Chip Peripheral Bus
OS	Original Size
OSA	Optimized Simulated Annealing
P2P	Point to Point
PB	Partial Bitstream

PBB	Partial Branch-and-Bound Algorithm
PCAP	Parallel Configuration Access Port
pdf	Posterior Density Function
PE	Processing Element
PFMAP	Particle Filter Mapping
PIP	Picture in Picture
PN	Particle Number
PNoC	Programmable Network on Chip
PPE	Power Processor Element
PR	Partial Reconfiguration
PRNG	Pseudo Random Number Generator
PRT	Pulse Repetition Time
PS	Partial Sum
RA	Reconfigurable Area
RAMPSOC	Runtime adaptive multi-processor system-on-chip
RCT	Router Configuration Topology
RL	Run Length
RPE	Reconfigurable (Image) Processing Element
RRR	Reconfiguration Repetition Rate
RSTA	Router-Switch Topology Architecture
ReCoSoC	Reconfigurable Communication-centric Systems on Chip
ReNoC	Reconfigurable Network on Chip
SA	Simulated Annealing
SIR	Sequential Importance Re-sampling
SoC	System On Chip
SPE	Synergestic Processor Element
SRAM	Static Random Access Memory
SS	Space Savings
OSA	Optimized Simulated Annealing
TGFF	Task Graphs for Free
TSV	Through Silicon Via

TTG	Task Traffic Graph
VHDL	Very High Speed Integrated Circuits Hardware Description Language
VIP	Virtual Point-to-Point
VOPD	Video Object Plane Decoder
VPU	Video Processing Unit
WC	Worst Case
WF	Wavefront

1. MOTIVATION

Today, most of the data intensive applications (e.g image, video, signal processing) are running on embedded multi-core architectures. On a multi-core architecture, there are several small cores which run mostly at lower frequencies. Increasing the number of cores leads to an increase in the number of messages communicated between them. This may end up with a reduced performance and increased energy consumption. Hence, the overall performance of a multi-core system is being affected not only by the computation load but also by the communications infrastructure.

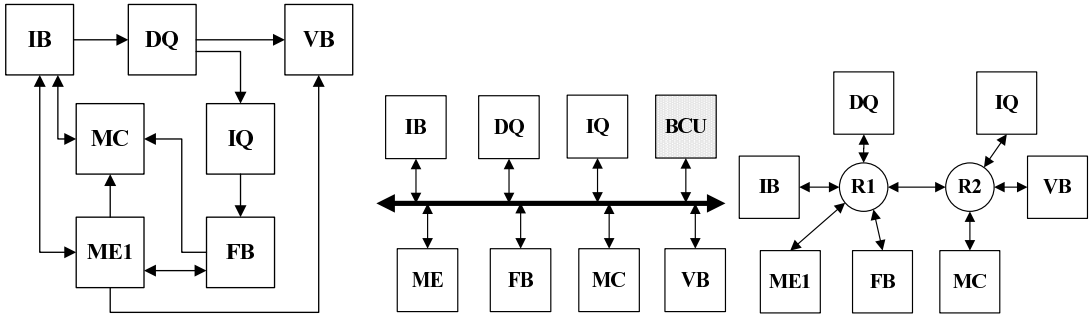


(a) Block Diagram (b) Connectivity Graph

Figure 1.1. MPEG2 application [3].

In Figure 1.1 embedded MPEG2 application is given. Here, while rectangular shapes represent the components of this embedded application (see Figure 1.1a), arrows between these components represent the communication interconnects of the application. Similarly, Figure 1.1b shows the connectivity of the components of this application.

There are, mainly, three on-chip communication architectures to connect the components of this embedded application. Figure 1.2 shows these possible architectures. Figure 1.2a can be the first choice, where each component is directly connected to each other via dedicated wires. Such an approach is called Point-to-Point (P2P). The second choice of on-chip communication architectures can be connection of these components via shared bus (see Figure 1.2b). As an alternative way, components can be connected



(a) Point-to-Point (b) Shared bus (c) NoC

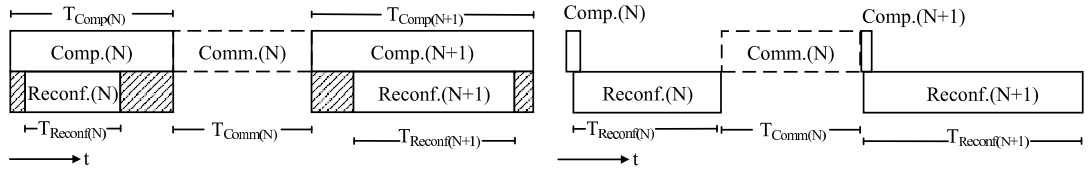
Figure 1.2. Possible on-chip communication architectures for MPEG2 application [3].

through routers in a Network-on-Chip (NoC) architecture as given in Figure 1.2c.

Here, the fastest communication architecture is the P2P architecture (see Figure 1.2a) where components are directly connected. Although P2P is the fastest solution among all these three communication architectures (see Figure 1.2), it turns out to be a power- and area-hungry solution as the number of cores increases. In shared bus communication architecture, component pairs communicate through a bus in different time slots (see Figure 1.2b). Here, the scheduling is done through a Bus Control Unit (BCU). Shared bus can be a good candidate if there are not too many connected components or cores. Like P2P, as the number of cores and message requests between these cores increase, shared bus approach becomes not scalable, as well [3].

In the last decade, the NoC has been proposed as an alternative to reduce power consumption and has been widely adopted by the SoC community. The third choice of on-chip communication architectures can be NoC architecture, where cores communicate through router elements (see Figure 1.2c). Although NoC architectures propose generic solutions and they are scalable, there are various parameters which affect the performance of NoC communication architecture directly or indirectly. Most important ones are mapping and routing algorithms, network topology, switching method, router architecture and link bandwidth. As the number of cores increase, routers become the bottleneck of NoC architectures because of the congestion and contention delays. In order to reduce negative impacts of routers, the first approach can be reduction the number of routers in a given NoC architecture. Hence, recent NoC studies [12–18]

try to suppress the drawbacks of routers used in both packet and circuit switched networks. The best way to relieve traffic density of routers and links can be reduction of contention and congestion delays. The second choice can be applying effective scheduling, mapping and routing processes to the NoC architectures. If the mapping and routing processes are not carried out properly, then improving other parameters will not improve the performance of NoC significantly. As a result of these, we firstly developed efficient mapping and routing algorithms for various NoC architectures in the scope of this thesis (see Chapters 9 and 10). By utilizing these efficient algorithms, we also propose dynamically reconfigurable point-to-point (DRP2P) interconnects for setting up direct connection between two communicating units before the communication starts [19]. In order to increase efficiency of DRP2P, we take the advantage of mapping and routing algorithms, which are mainly developed for NoC architectures: if the communicating nodes are located in an inefficient manner, the reconfiguration area might be larger than as expected. Increase in the reconfigurable area causes an increase in the reconfiguration time as well. Hence, a careful mapping strategy must be applied to the DRP2P for a successful design. In a similar manner, routing of interconnects must be carried out in a wise way such that interconnects do not exceed the boundaries of the pre-selected reconfigurable area. As a result of this, a smart routing must be applied to the DRP2P in order to keep the reconfigurable area as small as possible.



(a) Best Case Analysis (b) Worst Case Analysis

Figure 1.3. Computation, reconfiguration and communication periods in partially DRP2P interconnects.

As an alternative to the general purpose NoC architectures, DRP2P is neither point-to-point (P2P) nor Network-on-Chip (NoC); it stands between these two on-chip communication architectures. It is as fast as P2P and as scalable as NoC. DRP2P solves the scalability issue of P2P by setting up on-demand communication-specific links between cores. So, the occupied area and the total power consumption of communication architecture can be reduced significantly. Instead of using routers like in

NoC, we utilize partial reconfiguration ability of Field Programmable Gate Arrays (FPGAs) for routing data packets. Furthermore, DRP2P can work on any type of multi-core topologies. The only drawback in DRP2P is the reconfiguration latency. This drawback can be minimized when the reconfiguration of the communication links is achieved during the computation times of the cores as given in Figure 1.3a.

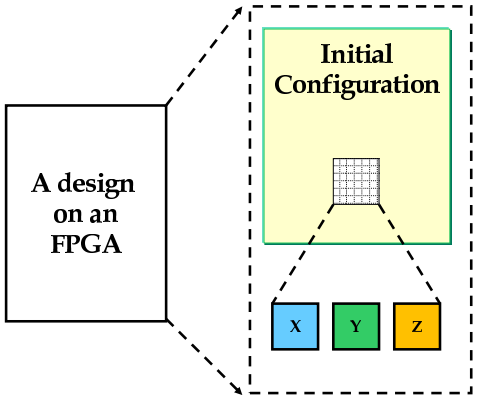


Figure 1.4. Pre-selected areas for partial reconfiguration.

In DRP2P, communication scenarios are downloaded to the FPGA one-by-one during the computation time of the cores. It is critical to understand that reconfiguration time plays a major role in assigning tasks to the cores and determining the number of communication scenarios: In Figure 1.3a, the communication architecture is established during the computation. Hence, reconfiguration time is not noticeable at all. This is the case, where DRP2P works as fast as P2P. However, in Figure 1.3b, the computation time is too short to set up the communication channel. In this case, the reconfiguration time introduces an overhead to the communication latency, hence degrades the efficiency of the P2P. This phenomenon dictates design constraints on self-reconfiguration engine:

- The reconfiguration engine must be dedicated and on-chip.
- It should be small to reduce area overhead.
- It should be as fast as possible so that the computing processors will not wait for the establishment of the communication channels
- It should have an on-chip cache to access the reconfiguration bitstreams quickly. Since on-chip memory is limited, the configuration bitstreams of communication scenarios should be small in size so as to increase the number of bitstreams on-

chip.

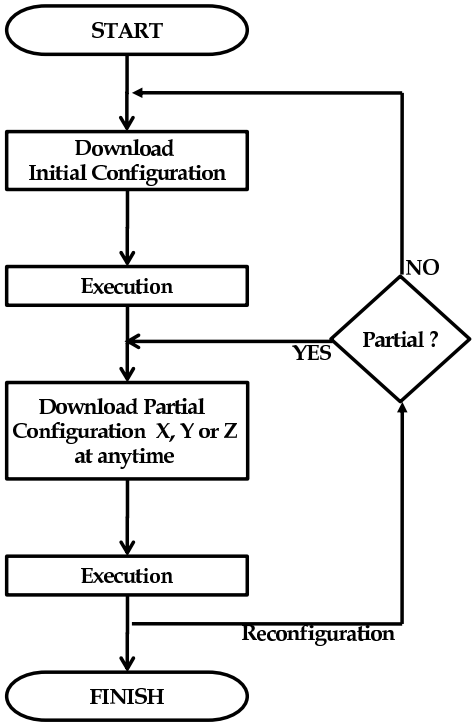


Figure 1.5. Partial reconfiguration flow.

As DRP2P takes the advantage of reconfiguration of interconnections between communicating nodes, the following paragraphs help in creating motivation to the partial reconfiguration concept.

Partial reconfiguration is the ability to reconfigure preselected areas of an FPGA anytime after its initial configuration while the design is operational (see Figure 1.4). By taking advantage of partial reconfiguration, hardware can be shared between various applications and upgraded remotely without rebooting and thus resource utilization can be increased [20].

Partial reconfiguration is allowed if the initial configuration is already loaded onto the target device. The partial reconfiguration flow is given in Figure 1.5. As it is obvious from this flow, partial bitstreams can be loaded onto the device only when they are required.

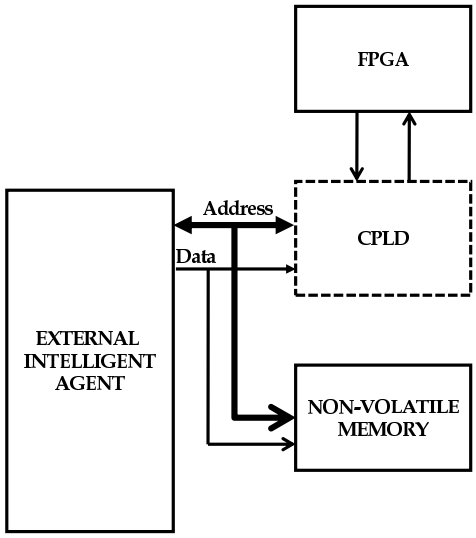


Figure 1.6. Dynamic partial reconfiguration.

In Figure 1.6, dynamic parital reconfiguration concept is given. In most cases a reconfigurable FPGA system consists of three main components: an external intelligent agent, some external (non-)volatile memory and a Complex Programmable Logic Device (CPLD). Such a reconfigurable FPGA system is described in detail in [10]. In some cases, systems may not require a CPLD if the used intelligent agent has a sufficient number of general purpose I/O (GPIO) pins. For these systems, the FPGA can be (re)configured directly by the intelligent agent [10].

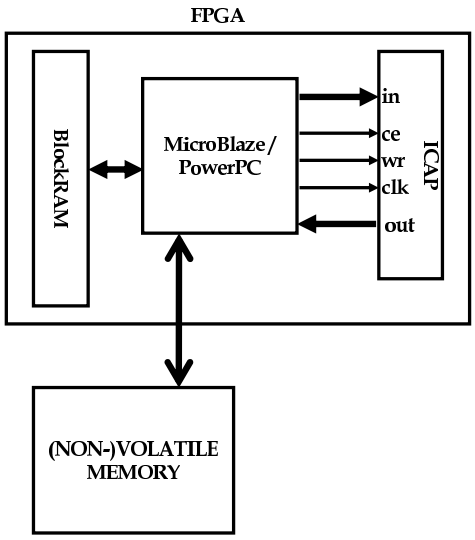


Figure 1.7. Dynamic partial self-reconfiguration.

As given in Figure 1.7, to be able to perform dynamic partial self-reconfiguration on an FPGA-based system, there should be either an internal configuration access port

or an equivalent port. Some FPGAs such as Spartan-3A(N), Virtex-II(Pro), Virtex-4, Virtex-5 from Xilinx, which have ICAP on their hardware, support self-reconfiguration without using an external intelligent agent. Hence, in the scope of this thesis, we designed three different on-chip self-reconfiguration cores, PCAP, cPCAP and c^2PCAP so as to achieve the runtime reconfiguraton of DRP2P interconnects as fast as possible. The latest and most efficient version of these reconfiguration engines, c^2PCAP, is used within the DRP2P study.

2. THESIS OUTCOME

2.1. Key Contributions

(i) *Design and development of self reconfiguration engines [8, 9, 19, 21]*: We propose three different dedicated reconfiguration engines in order to control reconfiguration flow for XILINX FPGAs. First version of these engines [8] has neither internal configuration nor bitstream compression and decompression capabilities. After noticing that configuration bitstreams can be compressed efficiently, bitstream compression and decompression properties are added to the previous engine. This modified version is our second reconfiguration engine [9]. In the third version of our reconfiguration engines [19], we added internal configuration capability at different configuration interface widths and a second level compression which is based not only on compression of a single bitstream but also on inter-bitstream similarities. All of these engines are very small in terms of hardware, they are portable, they can read streams from both off-chip and on-chip memories. None of them sacrifices from the decompression time. All these engines are dedicated for reconfiguration process and thus, they let the available processors on the target device without being interrupted by the reconfiguration process.

(ii) *An efficient and fast heuristic algorithm for NoC mapping problem [7]*: As the task to core mapping problem for NoCs is intractable, we propose a heuristic approach to solve this problem. The proposed algorithm can run in parallel and does not have a fixed time; algorithm's running time depends on the number of iterations given by the user. As the number of iterations of the algorithm increases, it tends to give better results. For this mapping problem, we make use of particle filters, systematic re-sampling and Dijkstra's shortest path algorithms.

(iii) *Particle Filtering based simultaneous mapping and routing algorithm for NoCs [22]*: Similar to the mapping problem for NoCs, the routing problem for NoCs is also intractable and dependent on the mapping quality. Hence, we propose a new heuristic algorithm, where we consider mapping and routing at the same time for a given input task graph and a NoC architecture topology. In this routing scheme,

we exploit particle filters, wave-front and Dijkstra's shortest path algorithms.

(iv) *Dynamically reconfigurable interconnects for multicore embedded systems [19]*: We propose a new communication architecture, which is neither point-to-point nor NoC but takes advantage of both approaches by reconfiguring the communication interconnects between cores. DRP2P is inspired from both P2P interconnects and NoC architecture. Since traffic flows between communicating nodes in most of the embedded applications are known in advance (e.g. at design time), DRP2P works as fast as P2P while reconfiguration is done at the time of computation. Thus, next communication scenario can be established by reconfiguration before communication starts. As a result, reconfiguration overhead can be minimized. DRP2P is as scalable as NoC; increasing the number of nodes or interconnects does not change the reconfiguration speed as long as they fit to pre-defined reconfigurable area. Unless the reconfigurable area changes, the size of partial bistream representing that area does not change. Thus, the corresponding reconfiguration time, which is directly proportional to the size of that bitstream, does not affect from the number of nodes or interconnects for a given embedded application.

2.2. Thesis Outline

Chapter 1 gives thesis motivation. In Chapter 2 key contributions are presented to the readers and also a summary of the whole thesis is given. In the following four chapters, we propose the reconfiguration engines, which are developed in the scope of the thesis. In Chapter 3, we give a brief introduction about dynamic partial self-reconfiguration on Xilinx FPGAs. Our first reconfiguration engine, named PCAP (Parallel Configuration Access Port) is explained in Chapter 4 in detail. In Chapter 5, the second reconfiguration engine, cPCAP (compressed Parallel Configuration Access Port), which has compression and decompression capabilities of bitstream, is examined widely. Chapter 6 gives the modified version of our cPCAP core (cPCAPv2) and its capabilities. Next, our most efficient reconfiguration engine, c^2PCAP (double compressed Parallel Configuration Access Port) and compression techniques are explained in Chapter 7. Afterwards, in the following four chapters, we show mapping and routing algorithms for NoC and our proposed solution DRP2P. In Chapter 8, we

give details about the topic of adaptive on-chip communication architectures of reconfigurable devices. The following chapter gives our developed mapping algorithm for NoC architectures and various related experimental results. Chapter 10 presents routing algorithm for NoC architectures. Then, in Chapter 11, we propose our dynamically reconfigurable interconnects solution. Later on, we present a conclusion describing the work done through the thesis in Chapter 12. The future opportunities in the scope of work given in the thesis is presented in Chapter 13. Finally, user manuals for reconfiguration engines, DRP2P, PFMAP and PFROUT are given in Chapters A, B and C respectively.

3. DYNAMIC PARTIAL SELF-RECONFIGURATION (DPSR) ON XILINX FPGAs

This chapter and the following three chapters present alternative approaches for dynamic partial self-reconfiguration that enables a Field Programmable Gate Array (FPGA) to reconfigure itself dynamically and partially through a parallel configuration access port under the control of a reconfiguration engine within the FPGA instead of using an embedded processor. The reconfiguration process is accomplished without an internal configuration access port(ICAP), which should be used either with MicroBlaze soft core or with PowerPC hard core using HWICAP core for the On-Chip Peripheral Bus (OPB) [10]. The three stand alone cores need neither HWICAP core nor the OPB interface.

The dynamic partial self-reconfiguration (DPSR) concept is the ability to change the configuration of part of an FPGA device by itself while other processes continue in the rest of the device. Normally FPGA devices can be reconfigured numerous times at runtime via an external intelligent agent such as a microprocessor, microcontroller, computer, or tester.

By exploiting the DPSR of an FPGA, a large design can fit into a smaller device. Even though the power dissipation during reconfiguration cannot be neglected, the total power consumption of the system can be decreased. It has already been shown that the system power or energy can be decreased by exploiting DPSR capability of FPGAs with using power reduction methods such as clock scaling, clock gating, reconfigurable hardware deactivation and removal, etc. [23–26]. There have been different works to achieve partial reconfiguration of Xilinx FPGAs. What seems to be disregarded thus far is the ability of self-reconfiguration of low cost FPGAs and the ability of self-reconfiguration on processor independent environments. So far, most of the works related to self-reconfiguration on Xilinx FPGAs are implemented with a custom core such as MicroBlaze, PowerPC, etc.

A pure Spartan-3 FPGA, which doesn't have any multiboot capabilities and Internal Configuration Access Port (ICAP), cannot be reconfigured without any additional external hardware. However, some other FPGA series such as Spartan-3A(N), Virtex-II(Pro), Virtex-4, Virtex-5 FPGA series have this ICAP module on their predesigned hardware architecture [10]. Although the Spartan-3 architecture is based on the Virtex-II and Virtex-II Pro architectures, the pure Spartan-3 family does not support the ICAP interface. In spite of the lack of an ICAP module on its architecture, dynamic reconfiguration is still supported in Spartan-3 via the external SelectMAP interface or JTAG, but not through the ICAP interface. Spartan-3 FPGAs support some of the dynamic partial reconfiguration capabilities, but with some limitations compared to Virtex devices.

Up to now, the lack of ICAP module on pure Spartan-3 FPGAs makes the DPSR impossible on these architectures without using any other additional external devices. ICAP is a functional subset of external parallel SelectMAP mode and is accessible internally via a user design. It allows the user design to control device reconfiguration at run-time. It becomes available after initial configuration is complete. That's why a component should be developed for pure Spartan-3 FPGAs, which acts as an ICAP and allows partial self-reconfiguration at run-time for Spartan-3 FPGA family. Partial reconfiguration is only possible through either serial JTAG interface or parallel slave SelectMAP mode. Since parallel slave SelectMAP interface has higher performance than the serial JTAG interface, the SelectMAP port is used in this thesis. Parallel SelectMAP port is used for either complete configuration or partial reconfiguration for applications, where the performance is the most important consideration.

The first self reconfiguration core developed within the scope of this thesis is PCAP (Parallel Configuration Access Port). The PCAP core needs only 324 slices, which is approximately 16% of a Spartan-3S200 FPGA. The dynamic partial self-reconfiguration via PCAP core works up to 50Mbyte/s. This approach has been implemented on a pure Spartan-3 FPGA from Xilinx, but it can also be used for any other FPGA architectures, such as Virtex-II(Pro), Virtex-4, Virtex-5, etc.

The extended version of the PCAP core is cPCAP, which is the first version of the self reconfiguration core operating on compressed bitstreams. Likewise PCAP, cPCAP uses BlockRAMs to store partial configuration bitstreams. BlockRAMs provide on-chip fast memory in FPGAs. However, the number of BlockRAMs is limited. Hence, cPCAP, maximizes the utilization of BlockRAMs by storing compressed partial bitstreams at initial configuration time and decompressing them during self reconfiguration. Due to compressed partial bitstream, more on-chip storage can be saved. Moreover, the reconfiguration clock speed of cPCAP is the same with that of PCAP in spite of the integrated decompression module. Because of being written entirely in VHDL, this cPCAP core is highly portable and can also be used for all other Xilinx FPGA architectures. Thus it is not necessary to use an external intelligent agent to control the partial reconfiguration flow. The cPCAP core with bitstream decompression module needs only 361 slices , which is approximately 18% of a Spartan-3S200 FPGA.

For Spartan-6 FPGAs, we have developed hard-macros and exploited the self-reconfiguration engine, compressed Parallel Configuration Access Port (cPCAP) [9], that we designed for Spartan-3. The modified cPCAP core with block RAM controller, bitstream decompressor unit and Internal Configuration Access Port (ICAP) Finite State Machine (FSM) occupies only about 82 of 6,822 slices (1.2% of whole device) on a Spartan-XC6SLX45 FPGA and it achieves the maximum theoretical reconfiguration speed of 200MB/s (ICAP, 16-bit at 100MHz) proposed by Xilinx.

The on-chip self-reconfiguration core cPCAP is improved further as c^2PCAP to maximize the utilization of on-chip memory further. This is achieved by applying joint compression on all partial bitstreams that may appear in the embedded system. For c^2PCAP core the combination of zero-run length coding and XOR method is proposed. Extreme compression ratios, like 99%, can be achieved when the similarities among bitstreams increase. Details of proposed compression method used in c^2PCAP core is explained in Section 7.2. Since the proposed compression method is not complex, it uses a very simple decompression module and as a result of this, the c^2PCAP core is extremely minimal. The decompression module of c^2PCAP core is explained in Section

7.4.

c^2PCAP core is designed for Xilinx FPGAs and can partially reconfigure the FPGA at the highest rate proposed by the manufacturer (up to 400MB/s for Virtex-4, 100MHz ICAP 32-bit; up to 200MB/s for Spartan-6, 100MHz ICAP 16-bit; up to 75MB/s for Spartan-3, 75MHz SelectMAP 8-bit). Moreover it is the smallest engine that is designed within this thesis: It occupies 208 slices for Virtex-4, 82 slices for Spartan-6 [21], 261 slices for Spartan-3.

3.1. Related Works on Self-Reconfigurable Systems

The fact that the pure Spartan-3 does not have such an internal configuration port, has rendered impossible the self-reconfiguration without using an external intelligent agent up to now apart from a few new studies, which are discussed below.

In [27], a soft ICAP, known as JCAP, has been developed in order to realize the self reconfiguration. As a reconfiguration interface they use serial JTAG interface which is very slow compared to parallel SelectMAP port. Though the ICAP on Virtex-II or Spartan3A devices have a reconfiguration speed 66MByte/s [10], JCAP only achieves a reconfiguration rate of 2Mbits/s. The reason of this huge performance difference between the ICAP and JCAP is the serial JTAG interface for JCAP.

Within the scope of the thesis, we have used parallel SelectMAP port instead of serial JTAG interface, thus we have developed a self-reconfigurable system on pure Spartan-3 series which should be at least 8 times faster than the developed system in [27]. Since a serial configuration method is used in [27], they achieved to send one bit per configuration clock cycle. However, our self reconfiguration cores use a parallel configuration method, hence we send 8 bits at each configuration clock cycle.

In [28], a self-reconfiguration system on pure Spartan-3 has been developed. They have solved the lack of ICAP on Spartan-3 FPGAs by adding an external loopback, therefore they have used a GPIO core on MicroBlaze and 11 external wires to accom-

plish the interface through SelectMAP port. In order to store initial configuration bitstream and generate configuration clock signal they have also used a XCF configuration flash PROM. Under the control of GPIO core of MicroBlaze they reconfigure the target FPGA through SelectMAP port. Though they have achieved a speed as in ICAP, they have used an external PROM to store initial configuration bitstream, MicroBlaze soft core to control the configuration flow and a TFTP server and onboard SDRAM to store the partial bitstreams. Although we have also used 11 external wires and the SelectMAP port as a reconfiguration interface, we have used BlockRAM to store partial bitstream, cPCAP core to control the reconfiguration flow. Since they designed a MicroBlaze-based system, they used 4198 slices of an Spartan-3S2000 FPGA. Applying such a system to small FPGAs (e.g. to a Spartan-3S200) is impossible. However, our cPCAP core is very small, which is 361 slices and only 18% of a Spartan-3S200. Thus, we have accomplished a processor-independent run-time reconfigurable system and presented a very new approach for storing compressed partial bitstreams on BlockRAM within the FPGA.

Both [27] and [28] process uncompressed partial bitstreams that are stored in external memory. However, cPCAP can process compressed partial bitstreams that are stored in BlockRAMs. This has two main advantages: (i) A BlockRAM can hold more compressed partial bitstreams than regular uncompressed partial bitstreams (ii) Access time to a partial bitstream in a BlockRAM is much shorter than the access time to a partial bitstream in an external memory. Decompression is realized in cPCAP during reconfiguration time without sacrificing from reconfiguration speed.

In [29], the functionality of JCAP core is extended with the readback feature for failure detection. Since readback is possible only through JTAG and SelectMAP interface on Spartan-3 devices and their JCAP core works only with JTAG interface, they prefer to use JTAG interface both for self-reconfiguration and readback processes. Although it is a time consuming process, the readback method is most common method for failure detection in configuration. The main drawback of this work is their slow reconfiguration and readback speed. For the reconfiguration of a module with the size 387 KBytes, they need 1.58 s which is really too long for reconfiguration time. So

their reconfiguration speed is 0.24 MB/s approximately. Apart from suffering from speed, the JCAP core neither offers a processer-independent platform for partial self-reconfiguration, nor does it apply partial bitstream compression.

In [30], an ICAP controller PLB ICAP, a rival to OPBHWICAP is proposed. This PLB ICAP controller is applied to Virtex-II Pro and Virtex-4 FPGAs from XILINX. Although the PLB ICAP controller attained higher reconfiguration throughput (295.4MByte/s) than OPBHWICAP's (5.07MByte/s), it can not still reach to maximum available throughput (400MByte/s) for Virtex-4 in 32-bit ICAP mode at 100MHz. Like JCAP, the PLB ICAP controller doesn't provide a processor-independency and bitstream compression.

In [31], Ming Liu *et al.* proposed to use DMA, Master burst (MST) and a dedicated Block RAM cache in order to reduce the reconfiguration time on a Virtex-4 FPGA. They made a comparison among different ICAP (32-bit, 100MHz) designs such as *opb_hwicap*, *xps_hwicap* with their proposed architectures *dma_hwicap*, *mst_hwicap* and *bram_hwicap*. As a maximum reconfiguration speed among these approaches, they have attained 371.4MB/s for *bram_hwicap*.

In [32], Koch *et al.* have demonstrated partial reconfigurable systems on Spartan-6 FPGAs. As the reconfiguration interface, they have used the ICAP with 16-bit Mode at 100MHz. To control the reconfiguration flow, they have used a host PC and the UART interface to access the ICAP. Since such a system does not reconfigure itself without any external engine, it is not in the class of self-reconfigurable systems. To put it clearly, a self reconfigurable system should have an internal agent/ processor/ FSM which controls the configuration flow of the FPGA device. When the reconfiguration process control engine is an external agent, then the whole system is not in the class of self-reconfigurable system but in the reconfigurable system. They have also mentioned maximum attainable reconfiguration speed (200MB/s) on Spartan-6 FPGAs, but not about the reconfiguration speed in their own system.

In [33], a self-reconfigurable system on a Virtex-4 FPGA is proposed. The pre-

ferred reconfiguration engine is the MicroBlaze soft core and the configuration interface is ICAP. The authors did not mention about the ICAP interface bit width that they have preferred in their study. Partial reconfiguration times that they obtained vary from 108ms to 460ms while partial bitstream (PB) sizes are in the range of 58KB to 244KB. Therefore the reconfiguration speed is about 0.54MB/s.

In [34] Hübner *et al.* proposed a fast dynamic and partial reconfiguration data path for Virtex-4 FPGAs. In their study, they have compared three different ICAP approaches (FSL-ICAP, XPS-ICAP, and pure ICAP) on the soft-core MicroBlaze and the hard core PowerPC. The maximum speed they have attained is 295.4MB/s with ICAP (32-bit, 100MHz) on PowerPC. For the PB repository, they have chosen external DDR2-SDRAM memory. This memory has different bus connections such as XCL bus and PLB bus over MPMC controller to the MicroBlaze and PowerPC respectively. The different speed values between these ICAP designs come from the bus type and the way they connect the bus to the DDR2-SDRAM memory.

François *et al.* have presented a fast ICAP controller on a Virtex-5 FPGA in [35]. In their study, they have reached the ICAP upper bound throughput of 800MB/s by overclocking the ICAP to 200MHz (32-bit mode). Actually there is no specific datasheet value for the ICAP maximum clock frequency, but it is clear that this value should never exceed the maximum clock frequency for any external configuration port (100MHz for serial/SelectMAP interface of Virtex-4 and Virtex-5 devices). Since exceeding this values may result in incorrect operation, we do not recommend overclocking the ICAP module. In addition to this, they have compared to two different ICAP approaches, *xps_hwicap* and *xps_hwicap* with direct memory access (DMA) at 100MHz. The maximum achieved ICAP bandwidth for two different approaches are 128MB/s and 174MB/s respectively.

In the framework for run-time reconfiguration represented in [36], a Virtex-II Pro FPGA XC2VP7 is reconfigured by using OPB-HWICAP under the control of PowerPC. In this study, the average reconfiguration time per frame is 3028 μs. However, copying the data to the BRAM into the configuration memory controller takes 2036 μs.

Therefore, the time taken up by the ICAP to perform the actual partial reconfiguration per frame is equal to 3028 μs-2036 μs = 992 μs. In XC2VP7, the frame length is 106 and each consisting of 32-bit words [10]. In 992 μs, Silva and Ferreira reconfigure 424 Bytes of configuration memory, therefore the reconfiguration speed is about 417KBytes/s.

A lot of studies related to dynamic partial reconfiguration property of reconfigurable architectures have been done in the literature. Especially, partial reconfiguration of FPGAs has an important role in such design techniques. There have been different works to utilize the partial reconfiguration ability of XILINX FPGAs [27–38]. We have summarized the most of these studies in Table 3.1.

Table 3.1. A summary of some previous works on self-reconfigurable systems.

Study	**Control Engine**	**Partial Bitstream (PB) Repository**	**Interface**	**Speed**	**Target Device**
[27]	MicroBlaze	Off-Chip	JTAG	2Mbits/s	Spartan-3
[28]	MicroBlaze	Off-Chip	SelectMAP	66MB/s	Spartan-3
[29]	MicroBlaze	Off-Chip	JTAG	0.24MB/s	Spartan-3
[30]	PLB ICAP	Off-Chip	ICAP	295.4MB/s	Virtex-II Pro and Virtex-4
[31]	MicroBlaze	Off-Chip	*dma_hwicap* *mst_hwicap* *bram_hwicap*	82.6MB/s 253.2MB/s 371.4MB/s	Virtex-4
[32]	host PC UART interface	Off-Chip	ICAP	200MB/s	Spartan-6
[33]	MicroBlaze	Off-Chip	ICAP	0.54MB/s	Virtex-4
[34]	MicroBlaze PowerPC	Off-Chip	ICAP FSL-ICAP XPS-ICAP	295.4MB/s	Virtex-4
[35]	MicroBlaze	Off-Chip	overclocked ICAP *xps_hwicap* *xps_hwicap*with DMA	800MB/s 128MB/s 174MB/s	Virtex-5
[36]	PowerPC	Off-Chip	ICAP	417KBytes/s	Virtex-II Pro
Our Study	c^2PCAP	On-Chip	SelectMAP or ICAP	300MB/s 400MB/s	All Xilinx FPGAs

All above mentioned designs either suffer from speed or do not offer a processor-independent self-reconfiguration platform for the low-cost state-of-the art FPGAs. In addition to these, overclocking of ICAP module of an FPGA is not recommended by the device vendor. To run safely, the speed limit of ICAP is 100MHz. Therefore, the maximum throughput that can be achieved is 400MB/s (32-bit mode, not supported in Spartan-6) theoretically. Since Spartan-6 supports reconfiguration through ICAP only

in 16-bit interface, the maximum theoretical reconfiguration speed through ICAP at 100MHz is 200MB/s for these devices. In our case, we propose a processor-independent self-reconfiguration platform for low-cost Spartan-6 FPGA with a safe throughput of 200MB/s (Spartan-XC6SLX45 ICAP, 16-bit mode, 100MHz).

Most of these studies process uncompressed partial bitstreams that are stored in external memory. However, c^2PCAP can process compressed partial bitstreams that are stored in BRAMs. This has two main advantages: (i) One BRAM can hold more compressed partial bitstreams than regular uncompressed partial bitstreams (ii) Access time to a partial bitstream in a BRAM is much shorter than the access time to a partial bitstream in an external memory. Decompression is realized in c^2PCAP during reconfiguration time without sacrificing from reconfiguration speed. In addition to the configuration bitstream's storage type, none of above-mentioned studies offers a processor-independent platform. All studies use MicroBlaze or PowerPC as their reconfiguration manager which results in large design sizes. However, our c^2PCAP core is very small, which is 261 slices. Another disadvantage of using MicroBlaze/PowerPC as a reconfiguration flow controller is that they have to postpone their computational jobs while they are busy with reconfiguration. This will slow down the operations which are executed by them. Therefore it has more sense to use a stand-alone core which is dedicated only for control of reconfiguration flow. Apart from these, c^2PCAP core use either parallel SelectMAP [10] or parallel ICAP [10] method in 8/16/32-bit modes, hence we send 8/16/32 bits at each configuration clock cycle.

Moreover, above mentioned designs either suffer from speed or do not offer a processor-independent self-reconfiguration platform for low-cost state-of-the art FPGAs. In addition to these, overclocking of ICAP module of an FPGA is not recommended by the device vendor. To run safely, the speed limit of ICAP is 100MHz. Therefore, the maximum throughput that can be achieved is 400MB/s (32-bit mode, not supported in Spartan-6) theoretically. Since Spartan-6 supports reconfiguration through ICAP only in 16-bit interface, the maximum theoretical reconfiguration speed through ICAP at 100MHz is 200MB/s. In our case, we propose a processor-independent self-reconfiguration platform for low-cost Spartan-6 FPGA with a safe throughput of

200MB/s (Spartan-XC6SLX45 ICAP, 16-bit mode, 100MHz).

3.2. Related Works on Configuration Compression Techniques

In the literature, there are studies which are directed for reducing the configuration size by proposing new mapping, placement, routing algorithms [39–44]. This step is done at the time of bit-stream generation or before it. However our method is bit-stream compression which is independent from mapping, placement, routing. Our technique is applied only to partial bitstreams generated by vendor tools such as bitgen from XILINX ISE.

There are various studies in the field of configuration compression and decompression techniques [45–50]. Common approaches are Run-length encoding, Huffman coding, Arithmetic coding, LZ based coding and their versions that are improved, extended or manipulated. Our study and some of example studies are summarized in Table 3.2.

Table 3.2. A summary of some previous studies on configuration compression techniques.

Study	**Technique**	**Approach**	**Decompression**	**Space Savings (up to)**	**Target Device**
[45]	Huffman, Arithmetic and LZ coding	Wildcard	complex	80%	Virtex
[46]	Adaptive LZW	intra-frame, inter-frame and inter-bitstream regularities	complex	45% for partial, 38% for complete	Virtex
[47]	RL encoding	Golomb code	simple	78%	N/A
[50]	LZ	dictionary	complex	41%	N/A
Our Study	RL encoding	XOR method	simple	99% for partial	All Xilinx FPGAs

In [45], Li and Hauck have researched different types of configuration compression techniques such as Huffman coding, Arithmetic coding and LZ coding for the Virtex devices. They have also developed very efficient algorithms including readback algorithm, frame reordering techniques and the wildcard approach. Although the algorithms they have developed seem to be very efficient, they are focused on only compression methods

for Virtex FPGAs and except the wildcard approach, all other algorithms are related to complete bitstreams of Virtex devices. Moreover, in the best case, the results of the wildcard approach seems to have 80% space saving.

Gu and Chen has developed an adaptive LZW algorithm for compression of Virtex partial bitstreams in [46]. By exploiting similarities among the hardware resources in a CLB array, they have taken the advantage of intra-frame, inter-frame and inter-bitstream regularities. In the best case, they achieved 45% compression ratio for partial bitstreams and approximately 38% compression ratio for complete bitstreams.

In [47], for the purpose of partial reconfiguration, bitstream extraction and merging techniques have been implemented. By extracting bitstreams between a base configuration and a combination of base configuration and a new small module, they obtain a bitstream that contains a high amount of zeros. Since they aimed to implement a platform independent partial bitstream manipulation, they have generated their partial bitstreams with their extraction method. However, the difference based partial bitstream generation approach from XILINX does the similar thing for the partial bitstream generation. Hence, we do not need such an extraction between two complete bitstreams. In order to compress bitstreams, they have exploited the run-length encoding with a modified Golomb code. Their results show that they achieved approximately 78% space saving.

In [50], Dandalis and Prasanna propose a novel configuration compression technique that can be applied to both complete and partial configuration. In their work they propose a compact memory representation for the dictionary used by any LZ based algorithm. Moreover they reduce the memory requirements of the dictionary by selectively decomposing strings in the dictionary. With their new approach they demonstrate up to 41% space saving in memory for configuration bitstreams.

In the literature, there are studies ([43, 44] and etc.) which are directed for compressing configurations. In these studies, the compressing step is done at the time of bit-stream generation or before it. However our method is bit-stream compression

which is independent from configuration compression and it can be directly used after configuration compression to improve the compression rate. Therefore, the subject configuration compression will not be explained in detail and not mentioned any more.

4. A RECONFIGURATION ENGINE FOR LOW-COST FPGAs: PARALLEL CONFIGURATION ACCESS PORT (PCAP)

This chapter describes custom soft PCAP core, which controls the partial reconfiguration flow through SelectMAP port and supplies configuration clock for reconfiguration. Because of being written entirely in VHDL, this PCAP core is highly portable and can also be used for all other Xilinx FPGA architectures. Thus it is not necessary to use an external intelligent agent to control the partial reconfiguration flow. As a result, using this PCAP reduces hardware cost and power consumption of a self reconfigurable system.

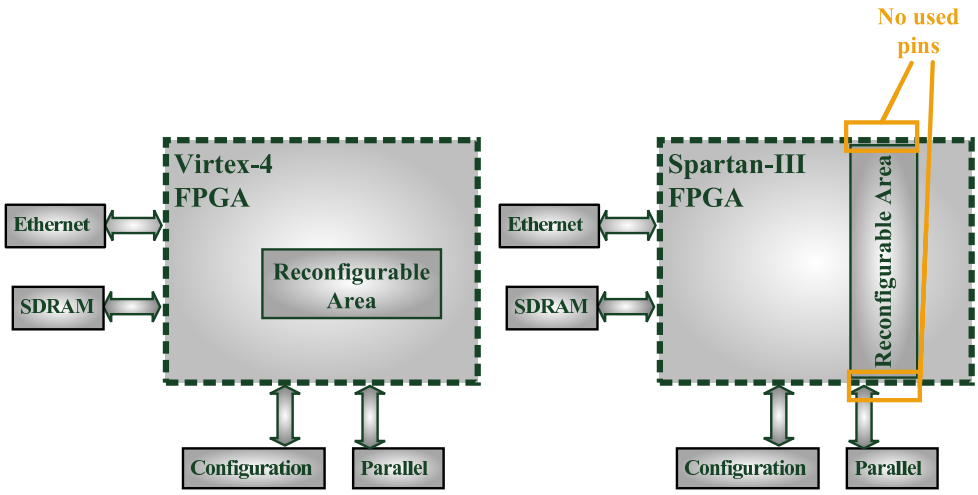


Figure 4.1. Possible reconfiguration areas in Virtex-4 and Spartan-3.

In Virtex-4 devices the minimal reconfiguration unit is frame, thus a Virtex-4 FPGA can be reconfigured two dimensional by issuing some previous commands. A frame on a Virtex-4 device can be reconfigured while the rest of the device continues its normal operation. If some bits of the new frame do not change in comparison to the older one, it is guaranteed that there will be no glitches on this bits during the reconfiguration. However, on a pure Spartan-3 FPGA the minimal reconfiguration unit is a whole CLB column as shown in Figure 4.1. It is not guaranteed that no glitches will happen during the reconfiguration process. As a result of this problem the reconfigurable area must be comprised of whole CLB columns, from top to bottom of

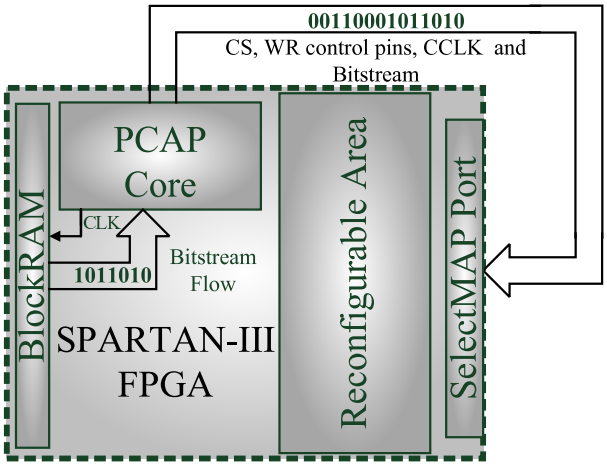


Figure 4.2. Hardware architecture of whole system with PCAP core.

the device. The only issue is here the FPGA pinout, there must be no used pins in the reconfigurable area [28].

4.1. PCAP Architecture

The target FPGA Spartan-3 in our system is configured by itself through its SelectMAP port under the control of PCAP defined within the FPGA. Through the parallel SelectMAP interface, which is 8 times faster than serial JTAG interface at the same frequency, the partial reconfiguration information is accepted by the target FPGA and the reconfiguration process is executed again by the same target FPGA, where this unique FPGA acts as not only a *slave* but also a *master* at the time of reconfiguration as shown in Figure 4.2.

As shown in Figure 4.3, in order to generate configuration clock frequency we have used a Digital Clock Manager (DCM) component, which is available within FPGA. Since we generate CCLK signal within FPGA, it acts as master, but at the same time we accept the CCLK signal through the SelectMAP interface into FPGA as if the signal comes from other intelligent agent, where the FPGA acts as slave. The source of CCLK is not important for the FPGA. The main point is that CCLK comes from outside. As a result of this, we should use the SelectMAP port in slave mode by setting the MODE pins as in Figure 4.3. The PCAP core reads a byte from the BlockRAM

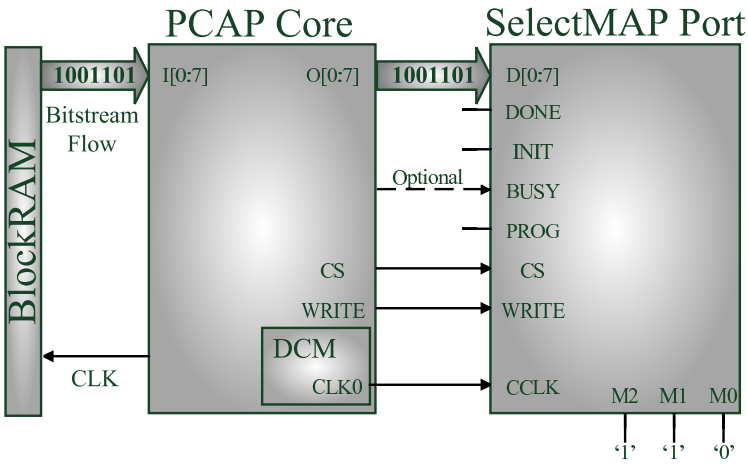


Figure 4.3. PCAP Core and SelectMAP interface.

at each clock cycle. Under the control of CS, WRITE, CCLK signals, this byte is sent to SelectMAP interface. When the CCLK speed is set to 50 Mhz, the reconfiguration speed is 50MByte/s. Note that the reconfiguration speed is independent from the size of partial bitstream. The BUSY signal is only used if the configuration clock frequency exceeds 50 Mhz. The PCAP core can be configured to operate up to 50 Mhz. We have not exceeded 50 Mhz yet, that's why we have not used BUSY signal.

The configuration flow of an FPGA for run-time reconfiguration via PCAP is shown in Figure 4.4. First of all, we have generated an initial configuration bitstream with empty BlockRAM. However, in this initial configuration file, the BlockRAM is allocated for partial bitstreams. After generating the initial configuration bitstream we have generated also the partial bitstreams with the help of bitgen -r flow [10]. Then we have converted the partial bitstream files to BlockRAM coefficient files. Afterwards we have loaded BlockRAM of Spartan-3 with these coefficient files, then we have generated the modified configuration bitstream that includes also partial configuration information.

The storage of partial reconfiguration bitstream requires also an additional external hardware for reconfigurable systems. In the most of reconfigurable systems the partial reconfiguration file is stored in an external non-volatile device. It can be read from there under the control of either an external intelligent agent or the FPGA itself, where FPGA acts as a *slave* and *master* respectively. Contrary to the standard meth-

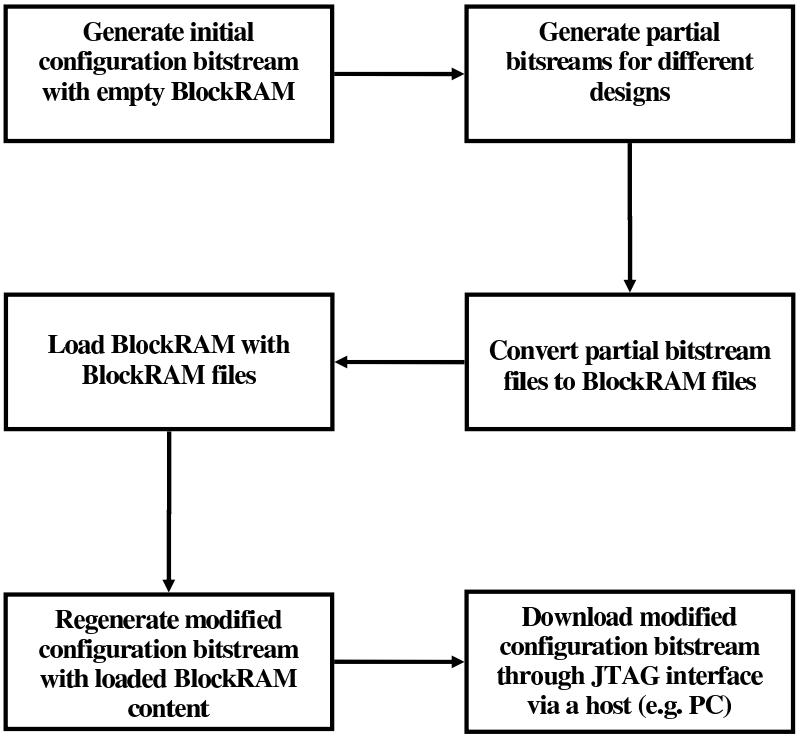


Figure 4.4. Configuration steps for PCAP core.

ods based on storing partial reconfiguration bitstream on external non-volatile devices, the partial reconfiguration bitstream in this thesis is stored in BlockRAM within the target FPGA. As a result of this new approach, there is no need to use an additional external device for storing partial reconfiguration bitstream for any system, which works continuously after the initial configuration.

4.1.1. File Converter

To store information in a BlockRAM there is a predefined file type with extension ".coe" that can be associated with the memory coefficients. After generating partial bitstream files, we have converted these files to a suitable form with ".coe" extensions using a file converter module, which is written in Java language, in Figure 4.5. Note that the number of partial bitstreams needs not to be equal to the number of BlockRAMs.

Figure 4.5. File conversion from partial bitstream file to BlockRAM coefficient file.

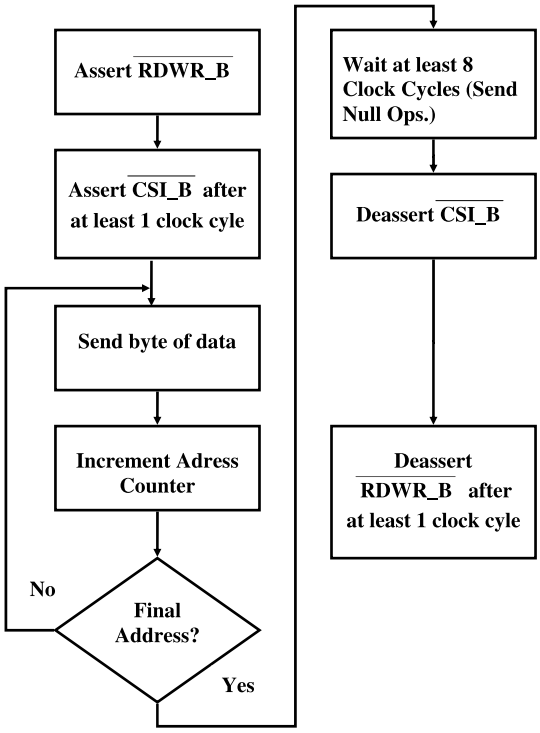


Figure 4.6. PCAP core configuration control flow diagram.

4.1.2. Dynamic Partial Self-Reconfiguration Flow

Since we have accomplished a dynamic partial self-reconfiguration through the SelectMAP port, the developed PCAP core behaves as if it is a mirror of SelectMAP port, as same in ICAP. The following flow in Figure 4.6 is very similar to "SelectMAP configuration Flow Diagram" in [10] except that PROG, INIT, DONE, BUSY pins are not taken into account.

The first three control signals PROG, INIT, DONE are only used during complete (re)configuration. BUSY signal is used if the configuration clock (CCLK) frequency is

greater than 50Mhz. Due to the fact that there is only a 50Mhz oscillator available on Spartan-3 Starter Board we have chosen the CCLK for our system exactly 50Mhz, thus we do not need to use BUSY indicator signal. Under some circumstances, such as using BUSY indicator signal where it is needed, the developed design can be clocked with any other frequency values, which are supported by Spartan-3 FPGA and its SelectMAP interface. We have experimented, that our PCAP core can run safely at all frequencies up to 50Mhz.

4.2. Case Study: Run-Time DCM Reconfiguration

The soft PCAP core, which is a pure VHDL code, has been synthesized on Spartan-3S200 Starter Kit Board. In this work we have reconfigured a clock output of Digital Clock Manager (DCM), which drives the complete system, at run-time. Such a DCM reconfiguration approach is described in detail in [23].

There are various applications, where clock frequency of a system should be changed at run-time. Clock scaling method in [23] is one of them, which is mostly used to decrease FPGA power consumption by changing clock frequency of different components of system at run-time. Changing data transmission speed of a system, and other typical applications include speed drives, inverters, computers and computer controlled equipment, deep well pumps, industrial machinery, ships, aircraft.

In this work, a 4-bit up-down counter is taken as an example. This counter either works with 5 Mhz or 50 Mhz, which is accomplished by run-time DCM reconfiguration via PCAP core. The size of each partial bitstream for DCM reconfiguration is 5 KByte . The reconfiguration speed is 50 MByte/s, which means that the DCM reconfiguration via PCAP core takes approximately 0.1 ms. This counter is used solely to show that such a reconfiguration approach is possible for other reconfigurable systems where the frequency of a system or a component of system can be changed at run-time without affecting anything else in complete system.

To be able to do reconfiguration we have firstly generated complete bitstream files

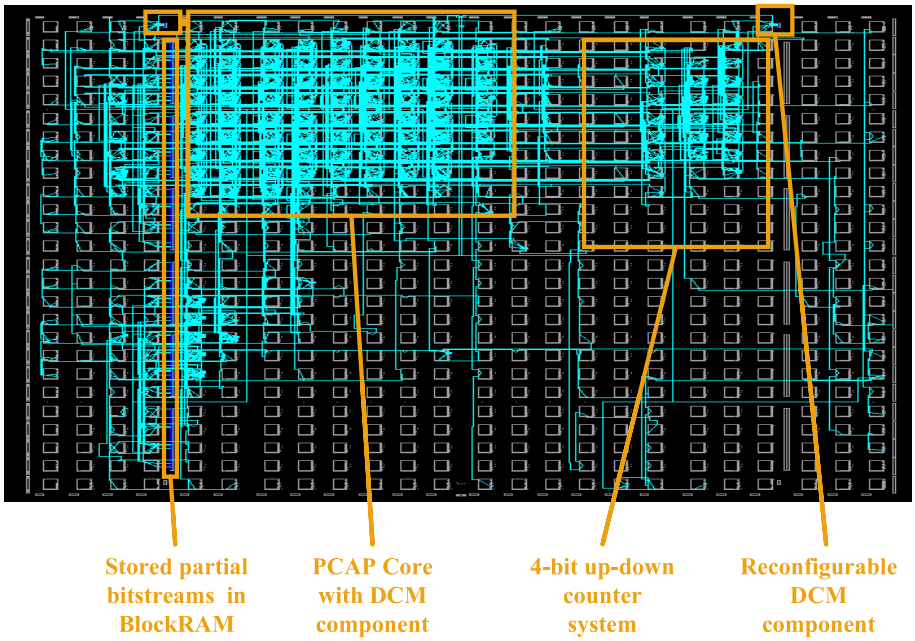


Figure 4.7. The complete system overview in FPGA Editor.

Table 4.1. FPGA resources.

	Occupied FPGA Slices	**Occupied BlockRAMs**	**Occupied DCMs**
PCAP Core	324 of 1920	-	1 of 4
Partial Bitstream Storage	-	6 of 12	-
Up-down Counter	41 of 1920	-	1 of 4
System	365 of 1920	6 of 12	2 of 4

and then with the help of bitgen tool two fully routed NCD (Native Circuit Description) file for each different frequency value. Contrary to the approach in [23] for generating partial bitstreams, the difference based approach is used in this work.

All above mentioned components of this system can be viewed in Figure 4.7, which gives an overview of complete system in the FPGA-Editor. To store partial bitstreams only 6 of 12 BlockRAMs are used as shown in Figure 4.7, and solely 365 of 1920 slices are occupied and also 2 of 4 DCMs are used in this project. However, just for PCAP core 324 of 1920 slices, which is approximately 16% of a Spartan-3S200 FPGA, and 1 of 4 DCM are used. These results are shown in detail in Table 4.1, which summarizes the implementation cost. Reconfiguring a DCM is actually reconfiguration

of the single column where DCM is. As a result, it can be obviously seen that PCAP core is much smaller than most of the soft controller.

5. A RECONFIGURATION ENGINE FOR COMPRESSED PARTIAL BITSTREAMS: COMPRESSED PARALLEL CONFIGURATION ACCESS PORT (cPCAP)

5.1. cPCAP Architecture

Likewise PCAP, the cPCAP core controls the reconfiguraton flow on the Spartan-3 FPGA through its SelectMAP port. Through the parallel SelectMAP interface, the compressed partial reconfiguration information is accepted and the reconfiguration process is executed again by the same target FPGA, where this unique FPGA acts as not only a *slave* but also a *master* at the time of reconfiguration as shown in Figure 5.1. While PCAP core processes uncompressed bitstreams, cPCAP core processes compressed bitstreams.

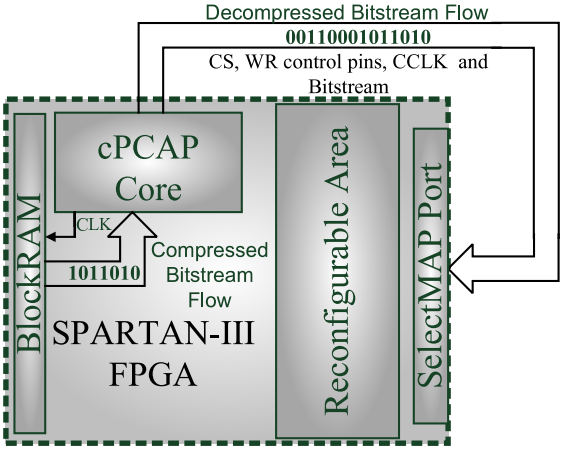


Figure 5.1. Hardware architecture of whole system with cPCAP core.

As shown in Figure 5.2, the cPCAP core reads compressed bitstream from the BlockRAM and decompresses the stream simultaneously, in order to send a byte to SelectMAP interface at each clock cycle. Since we achieve the decompression of bitstream information at the time of reconfiguration, we do not need any additional time for decompression. Therefore when the CCLK speed is set to 50 MHz, the reconfiguration speed is 50MByte/s exactly. Note that the reconfiguration speed is independent from the size of partial bitstream. The BUSY signal is only used if the configuration clock

frequency exceeds 50 MHz. In our study, the cPCAP core can be configured to operate up to 50 MHz. The configuration flow of an FPGA for run-time reconfiguration via cPCAP is shown in Figure 5.3. The uncompressed partial bitstreams are generated with the help of bitgen -r flow [10].

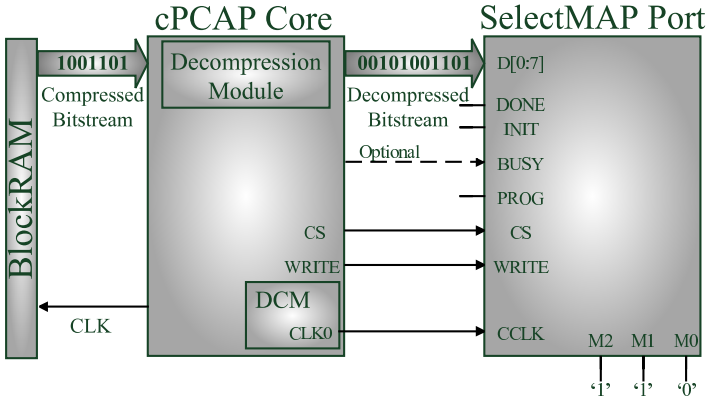


Figure 5.2. cPCAP Core and SelectMAP interface.

Initially, we generated an initial configuration bitstream with empty BlockRAM likewise for PCAP core. After generating appropriate partial bitstreams for different designs, we compressed and converted partial bitstreams. For the compression, we preferred to use run-length encoding [47] because of its simplicity. The compression methods will be mentioned in Section 7 in detail. After this step, we initialized BlockRAM contents with the compressed bitstream values. And, finally, we generated the modified configuration bitstream that includes also partial configuration information on the BlockRAMs as compressed.

5.1.1. File Compression and Conversion

There are a lot of different methods to store information in a BlockRAM. Using generic template from language templates in ISE is one of them. According to the bit-width and -depth of BlockRAM, there are various prepared pieces of code for BlockRAM available, which can be easily inserted into the BlockRAM HDL source file. Since we need to store partial bitstream bytes in BlockRAM, we have used one of these BlockRAM template.

After generating partial bitstream files, we have compressed these files and con-

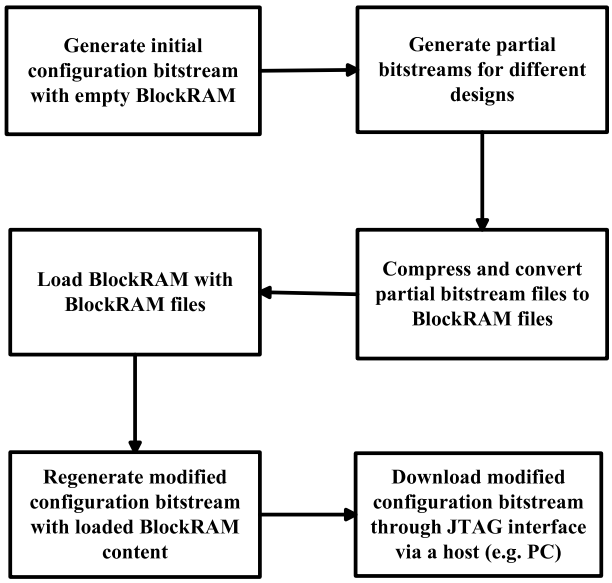


Figure 5.3. Configuration steps for cPCAP core.

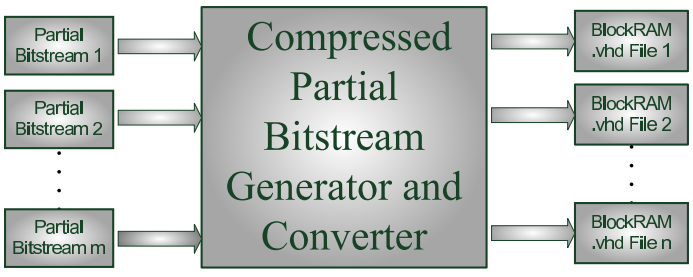


Figure 5.4. File conversion from partial bitstream file to BlockRAM coefficient file.

verted them to a suitable form with ".vhd" extensions using a file compression and converter module, which is written in Java language, in Figure 5.4. Note that the number of partial bitstreams needs not to be equal to the number of BlockRAMs.

Since both PCAP and cPCAPs send uncompressed bitstreams to the SelectMAP interface of the target FPGA, both have the same configuration control flow. Hence, the configuration control flow given in Figure 4.6 is also valid for cPCAP core.

5.2. PCAP vs. cPCAP

Table 5.1. Occupied resources for PCAP and cPCAP cores.

	PCAP (without compression)	**cPCAP (with compression)**
BlockRAM	6 of 12, %50	1 of 12, %8
Slices	365 of 1920, %19	324 of 1920, %16
DCM	1 of 4, %25	1 of 4, %25

Although both cores have many similarities, cPCAP outperforms PCAP core, because of its compressed bitstream processing capability. Comparison of PCAP and cPCAP is presented in Table 5.1. Here, we compare PCAP core and cPCAP core for the toy example given in Section 4.2. Although cPCAP core contains decompression module additionally, it occupies less slices than PCAP. The reason of this as follows: while PCAP uses BlockRAMs as IP cores, cPCAP exploits BlockRAM instance source codes.

After adding the decompression module to the cPCAP core, we have needed to use only 1 BlockRAM to store two different compressed partial bitstreams within the BlockRAM. With this compression approach we have accomplished at least 76% space saving approximately, where the compression ratio is actually based on the structure and size of the partial bitstream.

6. A RECONFIGURATION ENGINE WITH INTERNAL CONFIGURATION INTERFACE: EXTENDED VERSION OF COMPRESSED PARALLEL CONFIGURATION ACCESS PORT (cPCAPv2)

In this section, partial self-reconfiguration of Xilinx Spartan-6 FPGA with second version of cPCAP (cPCAPv2) is explained in detail. The cPCAPv2 core has more capabilities than pure cPCAP core:

- cPCAPv2 works with both SelectMAP interface and ICAP interface
- cPCAPv2 can run faster than 50MHz
- It is more suitable for new generation devices such as Spartan-6

There is still no partial reconfiguration tool support on low-cost FPGAs such as old-fashioned Spartan-3 and state-of-the-art Spartan-6 FPGA families by Xilinx. This forces the designers and engineers, who are using the partial reconfiguration capability of FPGAs, to use expensive families such as Virtex-4, Virtex-5 and Virtex-6 which are officially supported by partial reconfiguration (PR) software. Moreover, Xilinx still does not offer a portable, dedicated self-reconfiguration engine for all of the FPGAs. Self-reconfiguration is achieved with general-purpose processors such as MicroBlaze and PowerPC which are too overqualified for this purpose. In this part of the thesis, we propose a new self-reconfiguration mechanism for Spartan-6 FPGAs. This mechanism can be used to implement large and complex designs on small FPGAs as chip area can be dramatically reduced by exploiting DPSR feature for on-demand functionality loading and maximal utilization of the hardware.

To achieve a safe communication between the fixed area and the reconfigurable area, there should be routing resources, which are not affected during reconfiguration process. To do this, completely fixed and static hard bus macros (HBM) are preferred.

In this part of the thesis, we propose an extended version of our cPCAP [9] core which controls the partial reconfiguration flow through SelectMAP/ICAP port and supplies configuration clock for reconfiguration. The cPCAPv2 core with block RAM controller, bitstream decompressor unit and ICAP FSM manages the self-reconfiguration process through ICAP interface on a Spartan-6 FPGA. The details of design flow, generation of PBs, the way we store the PBs can be found in [9].

6.1. Configuration on Spartan-6 FPGAs

The Spartan-6 family is built on a 45-nm, 9-metal layer, dual-oxide process technology [51]. The Spartan-6 was marketed in 2009 as a low-cost solution for automotive, wireless communications, flat-panel display and video surveillance applications [51].

The partial self-reconfiguration on a Spartan-6 FPGA can be achieved through ICAP module of the device up to 100MHz with a 16-bit interface only. Spartan-6 FPGAs do not support 8-bit and 32-bit for ICAP. Therefore, the maximum self-reconfiguration speed that can be achieved on a Spartan-6 FPGA is 200MB/s. In addition to this, the external SelectMAP configuration interface can also be used for initial or partial reconfiguration. Note that some of Spartan-6 FPGAs (e.g. XC6SLX4 devices or devices using TQG144 or CPG196 packages.) do not offer the SelectMAP interface.

Opposite to the Spartan-3 FPGAs, Spartan-6 FPGAs offer the partial reconfiguration in a two dimensional fashion [32]. The atomic unit that can be reconfigured is a single frame within a clock region (in terms of CLB configuration). The configuration is in frames that covers 16 CLBs in the height. Therefore, for selecting reconfigurable area, the designer must take into account that the selected area should be within the same clock region, i.e. in the same 16-CLB-row. In Figure 6.1, three identical selected reconfigurable areas RA-1, RA-2, RA-3 are demonstrated on a Spartan-XC6SLX4 (the smallest member of Spartan-6 family) FPGA. The Spartan-XC6SLX4 is used only for demonstrative purposes, no actual design is implemented on this device. All implementations in this study are done on Spartan-XC6SLX45 FPGA. These areas have

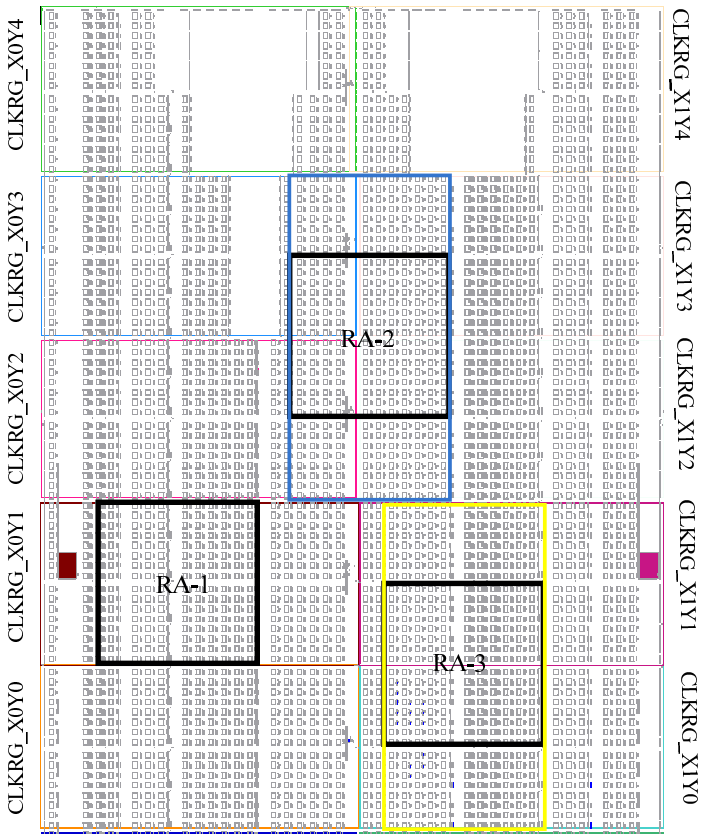


Figure 6.1. Selected reconfigurable areas on a Spartan-6 FPGA.

the same size but they are located in different portions of the FPGA. The RA-1 consists of 11CLB columns with 16-CLB-height in the same clock region. The RA-2 has also 11CLB columns with 16-CLB-height and is located in four different clock regions. The RA-3 has also 11CLB columns with 16-CLB-height and is located in two different clock regions. The generated PB for RA-1 will have 11CLB×(#Frames/CLB) frames information. However RA-2 will have 4CLB×(#Frames/CLB) frames in clock regions $X0\_Y2$ and $X0\_Y3$ separately and 7CLB×(#Frames/CLB) frames in clock regions $X1\_Y2$ and $X1\_Y3$ individually. So RA-2 will have 22CLB×(#Frames/CLB) frames information. In the same way, RA-3 will have 11CLB×(#Frames/CLB) frames in clock regions $X1\_Y0$ and $X1\_Y1$ individually which results in 22CLB×(#Frames/CLB) frames totally. Therefore, generated bitstream for RA-2 and RA-3 will be two times greater than RA-1, even if all portions does the same job. As a result, the reconfigurable area should fit into a clock region. Otherwise the number of frames included into the bitstream information will increase and the size of PB will be greater than expected. When the logic occupation in a clock region is not enough, then the neighbor

clock regions can also be utilized.

6.2. Self-Reconfiguration Platform

Our reconfiguration engine cPCAPv2 can use either SelectMAP or ICAP interface in any bit-width combination (i.e. 8, 16, 32-bit). We have adapted cPCAP core [9] to run on Spartan- XC6SLX45 FPGA with configuration interface ICAP 16-bit mode at 100MHz. Since cPCAPv2 core is entirely written in VHDL, it can be adapted to any other Xilinx FPGA to control the reconfiguration flow. As there is no SelectMAP interface on Spartan-XC6SLX4 FPGA devices (i.e. XC6SLX4, XC6SLX45, XC6SLX45T), we have used ICAP 16-bit mode with cPCAPv2 core.

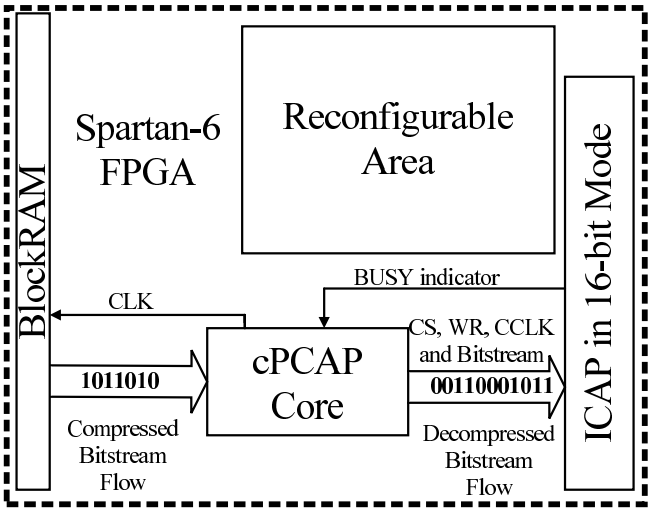


Figure 6.2. Hardware architecture of the system over internal configuration port (ICAP) on a Spartan-6 FPGA.

As shown in Figure 6.2, cPCAPv2 core reads compressed PBs from on-chip Block-RAM and decompress them on-the-fly at the time of reconfiguration. Therefore there is no downtime during decompression. As we use block RAMs to store compressed PBs and do not use any custom bus type, we have achieved the maximum throughput attainable by ICAP, 200MB/s (16-bit mode, 100MHz). The compression of PBs is done at design time. The method used for compression is the run-length encoding and does not require a complex decompression process. The architecture details of pure cPCAP and the compression/ decompression mechanisms can be found in [9].

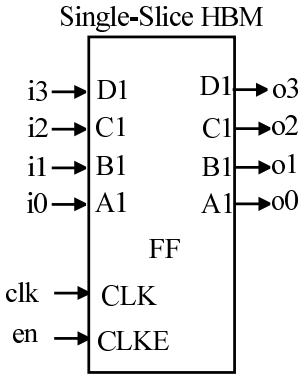


Figure 6.3. 4-bit single slice Spartan-6 HBM.

For the glitchless data flow between fixed area and reconfigurable area, we have used 4-bit slice based HBMs. Actually, there are no slice based HBMs offered by Xilinx for Spartan-6 FPGAs. However, there are some bus macros which are directly used for some target FPGA families (e.g. Virtex-5, Virtex-6) from Xilinx. Instead of designing a new bus macro we have manipulated and adapted Virtex-5 bus macros to the Spartan-6 bus macros. The 4-bit-width single slice HBM component is illustrated in Figure 6.3. In Figure 6.4, the internal structure and connections of our single slice synchronous Spartan-6 bus macro can be seen.

6.3. Test Results

Table 6.1. PB sizes and reconfiguration times for forward counter example.

	Original PB Size (Bytes)	**Compressed PB Size (Bytes)**	**# of BRAMs**	**Space Savings**	**Reconf. Time[μs]**
P1 (1Hz)	473	401	0.196	15.22%	2.365
P2 (5Hz)	473	401	0.196	15.22%	2.365
P3 (10Hz)	473	401	0.196	15.22%	2.365
P4 (20Hz)	473	401	0.196	15.22%	2.365
Total	1892	1604	0.783	15.22%	9.46

At first we have implemented our cPCAPv2 core for dynamically reconfiguring a few small sample circuits. The first one we have tested is a 4-bit forward counter with different speeds. To switch between different speeds we have reconfigured the clock

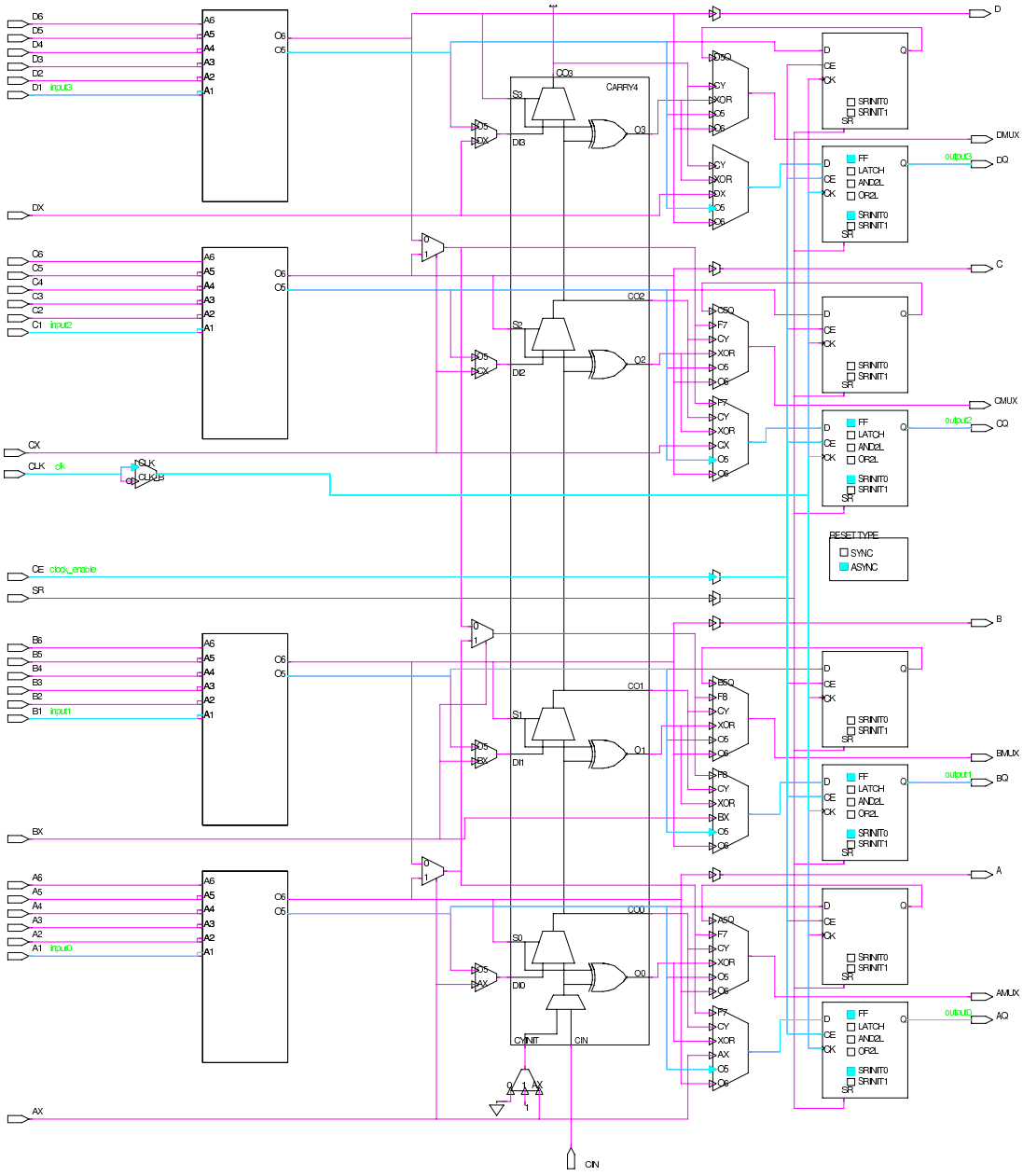


Figure 6.4. Inside of Spartan-6 slice based HBM.

input of the forward counter. According to this small example, there are 4 different speeds: 1Hz, 5Hz, 10Hz and 20Hz (to be observable by human eye) respectively. We have written a process to obtain such small clock periods. For the reconfiguration view, we have actually changed the trigger clock of this process. To be able to have different clock values, we have generated different clock values from digital clock manager (DCM) of Spartan-6 FPGA and changed these values at runtime by reconfiguring the counter circuit through ICAP under the control of cPCAPv2 core.

Table 6.2. PB sizes and reconfiguration times for HBM tester example.

	Original PB Size (Bytes)	**Compressed PB Size (Bytes)**	**# of BRAMs**	**Space Savings**	**Reconf. Time**[μs]
P1	603	349	0.170	42.12%	3.015
P2	603	348	0.169	42.29%	3.015
Total	1206	697	0.340	42.2%	6.03

There are various applications, where clock frequency of a system should be changed at run-time. Clock scaling method in [23] is one of them, which is mostly used to decrease FPGA power consumption by changing clock frequency of different components of system at run-time. Adjusting data transmission speed of a system, and other typical applications include speed drives, inverters, computers and computer controlled equipment, deep well pumps, industrial machinery, ships, aircraft. Actually, we are aware of the dynamic reconfiguration port (DRP) to reconfigure DCM outputs. But we just wanted to verify the correctness of our reconfiguration platform on a Spartan-6 FPGA. In this example, each PB is composed of only one frame. In Table 6.1, the original, compressed PB sizes and the reconfiguration times are available. In this example, we have achieved to put four different compressed PBs in a single block RAM. We have only used two Bytes (two consecutive 00 bytes)separators between each compressed PBs (total overhead is six Bytes). Note that, we have not excluded header information from original PB files generated by bitgen tool of Xilinx; before the compression process, we directly applied run length encoding compression technique to the PB files at design time.

The second small example is used to test the functionality of our HBMs. In this tiny example, we have tested two different small logical circuits, which gives different four-bit output values to the on-board LEDs. In this example, each PB is a composition of 2 frames. In Table 6.2, the original, compressed PB sizes and the reconfiguration times are available.

In addition to these tiny examples, the cPCAPv2 is also used to change the tasks (modes) of a Reconfigurable Image Processing Element (RPE). This case study can be found in [21].

7. A RECONFIGURATION ENGINE UTILIZING INTER-BITSTREAM COMPRESSION: DOUBLE COMPRESSED PARALLEL CONFIGURATION ACCESS PORT (c^{2}PCAP)

Our most improved reconfiguration engine is called c^2PCAP (double times compressed Parallel Configuration Access Port). The c^2PCAP core behaves as if it is a mirror of SelectMAP port, as same in ICAP. The configuration control flow in this work is very similar to "SelectMAP configuration Flow Diagram" in [10] except that PROG, INIT, DONE pins are not taken into account in our study. This control flow has been already illustrated in detail in Figure 4.6 in Section 4.1.2.

7.1. c^{2}PCAP Configuration Flow

The configuration flow for c^{2}PCAP is given in Figure 7.1. In this Figure, numbers over the boxes represent steps for the configuration under the control of c^{2}PCAP.

These steps are given as follows:

(i) Generate all necessary partial bitstream files with bitgen -r flow
(ii) Compress bitstreams by run-length encoding and their similarities
(iii) Select optimum partial bitstream set with the smallest size
(iv) Store bitstreams either on off-chip or on-chip BlockRAM
(v) Load FPGA with initial configuration
(vi) If partial reconfiguration is requested, check whether partal bitstream is already available on FPGA
(vii) If the desired PB is not available on FPGA, prefetch it from the off-chip memory
(viii) If it is either already available or pre-fetched decompress it
(ix) As decompression process starts, start reconfiguration also immediately

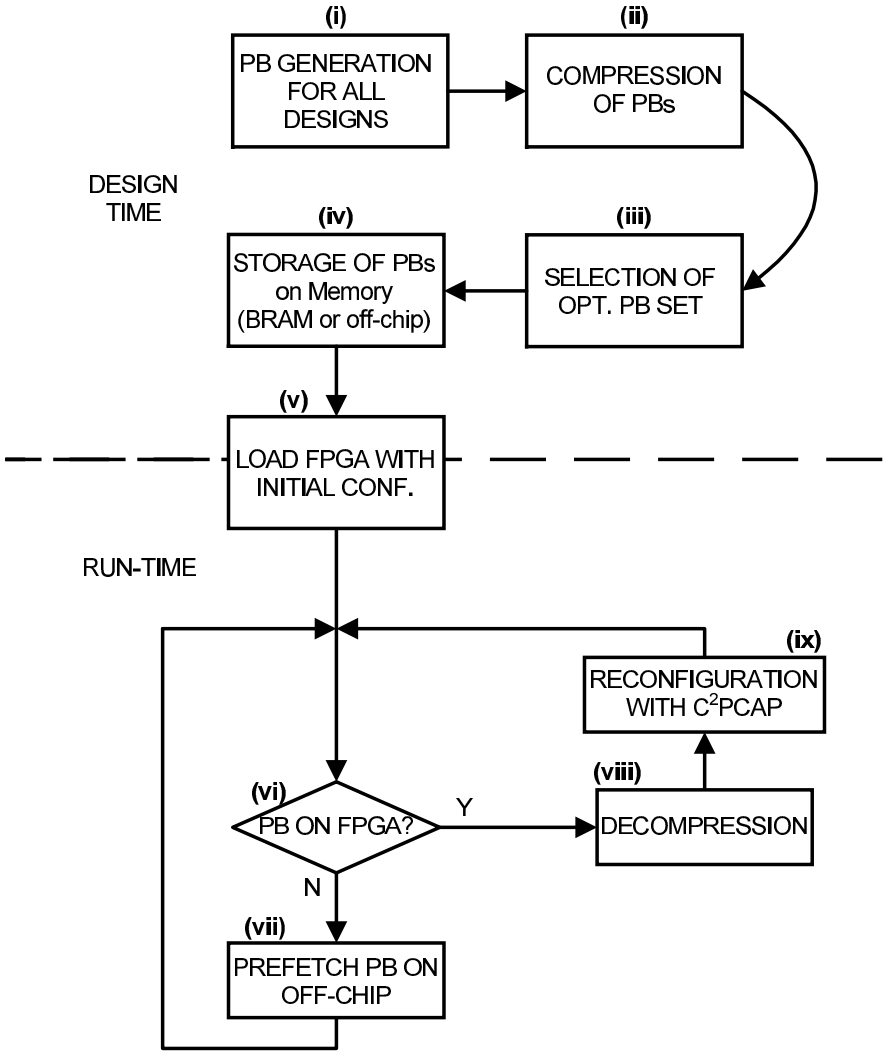


Figure 7.1. Configuration steps for c^{2}PCAP.

Abovementioned steps will be examined in detail in the following sections. Our proposed c^2PCAP is designed for decompressing the compressed bitstreams stored in the on-chip BlockRAM of the Xilinx FPGAs.

7.2. Compression

7.2.1. Partial Bitstream Similarity Extraction

Let p_i and p_j denote two partial bitstreams that successively reconfigures a region on the FPGA for DRP2P interconnects. Since this region has to be determined during design time, the length of both partial bitstreams are the same. Then we can extract similarities between two partial bitstreams by a simple XOR operation.

$$p_{ij} = p_{ji} = p_i \oplus p_j \tag{7.1}$$

Let p_{ij} be named as the joint bitstream of p_i and p_j. Obviously, as the similarities between p_i and p_j increases, the number of zero entries in p_{ij} increases. If there are N different communication scenarios in an application, then there are $C\left(N, 2\right)$ joint bitstreams. Note that XOR operation also helps in generating partial bitstream from another one by using the joint bitstream:

$$p_j = p_i \oplus p_{ij} \tag{7.2}$$

7.2.2. Compression of Partial Bitstreams

We introduce a zero run length coding technique to compress both types of bitstreams. In this technique, each byte in the bitstream is checked whether it is 0 or not. If it is not zero, it is directly written to the BRAM file. Otherwise, firstly the number of zero bytes is determined. Then the byte count is written to the BRAM file and the parity bit of the related memory location is set. Let l_i and l_{ij} represent the lengths of compressed partial and joint bitstreams, namely p_i^c and p_{ij}^c respectively. Note that the length of compressed bitstreams is usually smaller than the related partial and joint bitstreams.

7.2.3. Selection of Optimal Compressed Bitstream Set

There are $C(N,2)+N$ compressed bitstreams for N communication scenarios. We define I^s as the optimal set with N compressed bitstreams and this set can be used in the generation of all partial bitstreams by using Equation 7.2 after decompression. Therefore it is obvious that I^s must contain at least one compressed partial bitstream. The other members of I^s can be either compressed partial or compressed joint bitstreams. Hence, our aim is to obtain I^s such that the sum of the lengths of the selected compressed bitstreams is minimum so as to use as small BRAM area as possible. In this case, our problem turns out to be a subset sum problem which is known to be NP-complete. The fastest known exact algorithms utilizing dynamic programming whose worst case execution time would be around $O\left(K^2 2^K\right)$, where $K = C(N,2)+N$ if we had used it for the solution of our problem. However we propose a faster exact algorithm that has the time complexity of $O\left(N^2 2^N\right)$, because the Algorithm 7.2 is designed to omit infeasible solutions in the design exploration space. It operates on the indices of compressed bitstreams and initially assumes that the set of compressed partial bitstreams is the best solution (lines 1, 2). Then it exhaustively searches all possible solutions. In each iteration, there are $m \geq 1$ compressed partial bitstreams (lines 3-6). The remaining members are compressed joint bitstreams which are generated from G_x (line 6) and R (line 7). G_x is the subset of I^s and it contains partial bitstreams p_i that have to be stored in BRAM. R contains the indices that are in I^s

but not in G_x. Compressed joint bitstreams p_{ij} will be generated from the compressed bitstreams whose first index is from G_x and second index is from R. Initially the cost of $p'_i s$ ($i \in G_x$) is calculated (line 8). Then the costs of joint bitstreams which use $i \in G_x$ as the first index are evaluated one by one.The index pair (i, j) for compressed joint bitstreams is selected in a way to minimize the storage cost in BRAM (lines 8-14).

A sample run of our algorithm is available in Figure 7.3. The sizes of compressed partial and joint bitstreams are shown in tables. The rule for filling the tables is defined with the following equations:

$$\begin{aligned} Table(p_i, p_i) &= l_i \\ Table(p_i, p_j) &= Table(p_j, p_i) = l_{ij} \end{aligned} \tag{7.3}$$

There are four different communication scenarios in this example. Therefore, there are four partial, ten joint bitstreams. Each row includes sizes of one compressed partial bitstream and $N-1$ compressed joint bitstreams that can be used in the generation of all other partial bitstreams with the original bitstream in that row. For example, $Table(p_2, p_2)$ shows the bit-length of p_2^c and $Table(p_2, p_1), Table(p_2, p_3)$ and $Table(p_2, p_4)$ show the bit-lengths of p_{21}^c $(= p_{12}^c), p_{23}^c$ and p_{24}^c respectively. Hence, p_1^c, p_3^c and p_4^c can be generated as follows: $p_1 = d(p_2^c) \oplus d(p_{12}^c)$, $p_3 = d(p_2^c) \oplus d(p_{23}^c)$ and $p_4 = d(p_2^c) \oplus d(p_{24}^c)$. Here, $d(.)$ is the decompression function explained in Section 7.4. The best intermediate solutions are identified with shaded cells and bold entries in each table. At first, the initial cost (line 2 in Figure 7.2) is calculated as in the first table in Figure 7.3. The second table calculate the costs of partial bitstreams set for $m = 1$ (the first iteration of "while" loop), where G_x contains only a single partial bitstream. In tables 3 to 8, where G_x contains two different original bitstreams, the costs of partial bitstreams set for $m = 2$ (the second iteration of "while" loop) are calculated. In tables 9 to 12, where G_x contains three different original bitstreams, the costs of partial bitstreams set for $m = 3$ (the third and last iteration of "while"

```
/*Pick the smallest partial bitstream set*/
1: I^s = I = {1, 2, ..., N − 1, N}
2: cost=∑_{i∈N} l_i;
3: m=1;
4: while (m < N) do
5:    K_m = {the set of all subsets with m members from I}
6:    for each G_x ∈ K_m, 1 ≤ x ≤ C(N, m) do
7:       R = I − G_x;
8:       sum = ∑_{i∈G_x} l_i;
9:       for each j ∈ R do
10:         cost_j = min_{i∈G_x} l_ij
11:         i* = {the i ∈ G_x with cost_j};
12:         if sum + cost_j ≥ cost then
13:            break;
14:         end if
15:         sum+ = cost_j;
16:         S = S ∪ {i*j};
17:      end for
18:      if sum < cost then
19:         cost = sum;
20:         I^s = G_x ∪ S;
21:      end if
22:   end for
23:   m++;
24: end while
```

Figure 7.2. Picking the smallest partial bitstream set.

Table 1: Initial Cost

X	p_1	p_2	p_3	p_4	sum
p_1	**2163**	-	-	-	-
p_2	-	**2565**	-	-	-
p_3	-	-	**2510**	-	-
p_4	-	-	-	**2442**	-
					9680

G_x=(1,2,3,4}), S= (),
Sum = 9680, Cost = 9680, I^S =(1, 2, 3, 4)

Table 2: m=1, iterations 1-4

X	p_1	p_2	p_3	p_4	sum
p_1	2163	3099	3019	2926	11207
p_2	3099	2565	1742	2466	9872
p_3	**3019**	**1742**	**2510**	**2129**	**9400**
p_4	2926	2466	2129	2442	9963

G_x=(3), S= ({13}, {23}, {34}),
Sum = 9400, Cost = 9400,
I^S =(3, {13}, {23}, {34})

Table 3: m=2, iteration 1

X	p_3	p_4
p_1	3019	2926
P_2	**1742**	**2466**

G_x=(1, 2), S= ({23}, {24}),
Sum = 8936, Cost = 8936,
I^S =(1, 2, {23}, {24})

Table 4: m=2, iteration 2

X	p_2	p_4
p_1	3099	2926
p_3	**1742**	**2129**

G_x=(1, 3), S= ({23}, {34}),
Sum = 8544, Cost = 8544,
I^S =(1, 3, {23}, {34})

Table 5: m=2, iteration 3

X	p_2	p_3
p_1	3099	3019
p_4	**2466**	**2129**

G_x=(1, 4), S= ({24}, {34}),
Sum = 9200, Cost = 8544,
I^S =(1, 3, {23}, {34})

Table 6: m=2, iteration 4

X	p_1	p_4
p_2	3099	2466
p_3	**3019**	**2129**

G_x=(2, 3), S= ({13}, {34}),
Sum = 10223, Cost = 8544,
I^S =(1, 3, {23}, {34})

Table 7: m=2, iteration 5

X	p_1	p_3
p_2	3099	**1742**
p_4	**2926**	2129

G_x=(2, 4), S= ({14}, {23}),
Sum = 9675, Cost = 8544,
I^S =(1, 3, {23}, {34})

Table 8: m=2, iteration 6

X	p_1	p_2
p_3	3019	**1742**
p_4	**2926**	2466

G_x=(3, 4), S= ({14}, {23}),
Sum = 9620, Cost = 8544,
I^S =(1, 3, {23}, {34})

Table 9: m=3, iteration 1

X	p_4
p_1	2926
p_2	2466
p_3	**2129**

G_x=(1, 2, 3), S= ({34}),
Sum = 9367, Cost = 8544,
I^S =(1, 3, {23}, {34})

Table 10: m=3, iteration 2

X	p_3
p_1	3019
p_2	**1742**
p_4	2129

G_x=(1, 2, 4), S= ({23}),
Sum = 8912, Cost = 8544,
I^S =(1, 3, {23}, {34})

Table 11: m=3, iteration 3

X	p_2
p_1	3099
p_3	**1742**
p_4	2466

G_x=(1, 3, 4), S=({23}),
Sum = 8857, Cost = 8544,
I^S =(1, 3, {23}, {34})

Table 12: m=3, iteration 4

X	p_1
p_2	3099
p_3	3019
p_4	**2926**

G_x=(2, 3, 4), S= ({14}),
Sum = 10443, **Cost = 8544,**
I^S =(1, 3, {23}, {34})

Figure 7.3. Steps of the bitstream set selection algorithm (Figure 7.2) for an eight-core implementation on a Virtex-4 FPGA. (X = XOR, if i ≠ j; X = AND, if i = j).

loop) are calculated. In this example, the solution (I^s = 1,3,{23},{34}) is found in the fourth table ($m = 2, Iteration = 2$). Note that, the inner "foreach" loop (lines 9-17) of Figure 7.2 finds the minimum value in each column. The iterations for each m value are relevant to outer "foreach" loop (lines 6-22) of Figure 7.2.

7.2.4. Storage of Partial Bitstreams on Memory

After compression and selection of optimum set processes, partial bitstream of each communication scenario is stored as initial values for BlockRAM or off-chip RAM.

7.3. c²PCAP Architecture

Our most improved reconfiguration engine, c²PCAP has more capabilities than both PCAP and cPCAP cores:

- Compression ratio of c²PCAP core is much more than PCAP and cPCAP cores
- c²PCAP core can run up to the reconfiguration speed limits of the target devices

- It has the ability to work both SelectMAP and ICAP interfaces
- For ICAP interfaces, it does not require any external wires
- It exploits similarities of partial bitstreams

All processes including "Decompression", which are being executed at run-time, are done under the control of c^2PCAP core.

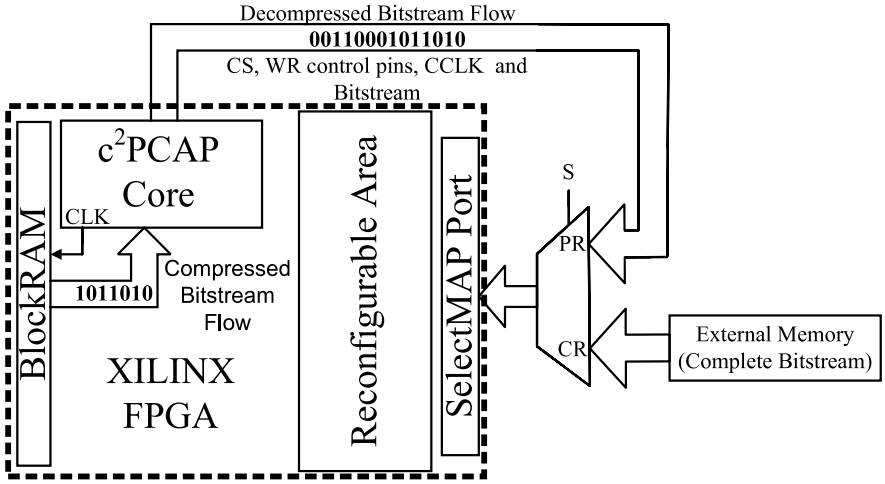


Figure 7.4. Hardware architecture of the system over external configuration port.

An FPGA from XILINX configures itself through its SelectMAP port under the control of c^2PCAP. Through the parallel SelectMAP interface, the partial reconfiguration information is accepted and the reconfiguration process is executed again by the same target FPGA, where this unique FPGA acts as not only a *slave* but also a *master* at the time of reconfiguration as shown in Figure 7.4. Note that the SelectMAP interface is not only dedicated for partial reconfiguration (PR), it might also be used in other designs for external configuration memory (CR:Complete (Re)Configuration, S is used for switching between PR and CR).

The similar structure is also used for devices, which have ICAP modules like Spartan-6, Virtex-4 FPGA. So, through its ICAP interface, the Xilinx FPGA configures itself under the control of c^2PCAP core. As shown in Figure 7.5, it is noted that the supported bit-width of ICAP for configuration varies considerably from architecture to architecture: 8-bit for Virtex-II(Pro), 16-bit for Spartan-6, 8/32-bit for Virtex-4, 8/16/32-bit for both Virtex-5 and Virtex-6 [10].

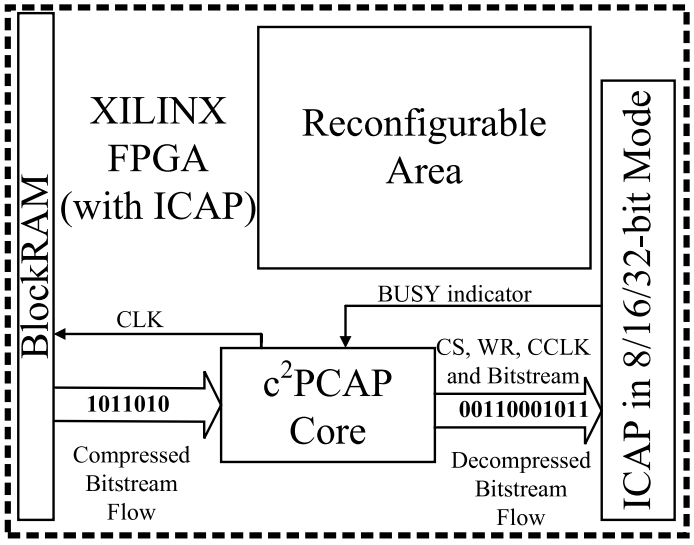


Figure 7.5. Hardware architecture of the system over internal configuration port.

As shown in Figure 7.6, the configuration clock (CCLK) is internally generated by DCM. Since FPGA acts as slave during reconfiguration, the source of CCLK is not important for the FPGA. The c^2PCAP core reads a byte from the BRAM at each clock cycle. Under the control of CS/CE, WRITE, CCLK signals, this byte is sent to SelectMAP for Spartan-3 and to ICAP interface for Virtex-4.

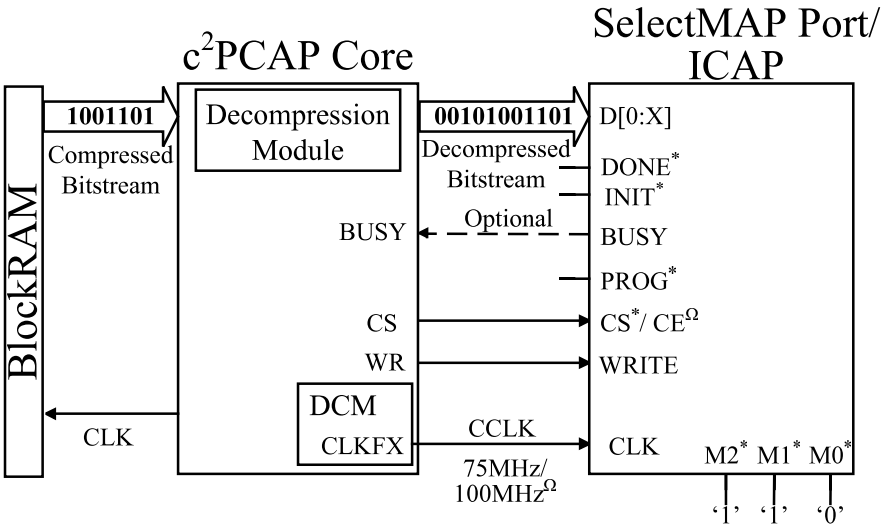


Figure 7.6. c^2PCAP core and SelectMAP/ ICAP interfaces(X:7/15/31, *: only for SelectMAP, $^{\Omega}$:only for ICAP).

The bitstream information, which is accepted from BRAM, is compressed. Since the decompression of bitstream information is achieved at the time of reconfiguration, there is no need any additional time for decompression. Therefore when the CCLK

speed is set to 75 MHz, the reconfiguration speed is 75MB/s. Note that the reconfiguration speed is independent from the size of partial bitstream. The BUSY signal is only used if the configuration clock frequency exceeds 50 MHz. In this study, the CCLK for c^2PCAP core can be configured to operate up to 75 MHz for Spartan-3 and 100 MHz for Spartan-6, Virtex-4. As a result we reached 75MB/s (75MHz, 8-bit SelectMAP) on Spartan-3, 200MB/s (100MHz, 16-bit ICAP) on Spartan-6, 300MB/s (75MHz, 32-bit ICAP) for compressed and 400MB/s (100MHz, 32-bit ICAP) for uncompressed partial bitstreams on Virtex-4.

7.4. Decompression

It is done by hardware and consists of three components as explained in the following subsections.

7.4.1. Look up table

It is a BRAM unit and contains the indices of partial bitstreams which are used in the generation of N communication scenarios. Each communication instance is represented by two address fields in the table. Fields I and II represent the address entries for the compressed partial and the compressed joint bitstreams respectively. If a communication instance can be implemented solely by a partial bitstream, then its corresponding joint bitstream entry is null. An example is shown in Table 7.1 for N=4. As an example, we may use the optimum set, p_1^c,p_3^c,p_{23}^c,p_{34}^c, which is found in Figure 7.3. For this example, in the look up table, we have address entry values of p_1^c,p_3^c,p_3^c,p_3^c in Field I and 0,p_{23}^c,0,p_{34}^c in Field II respectively.

7.4.2. Decompressor

If the parity of the byte in the compressed bitstream in BRAM is 1, then as many zeroes are generated as the byte value. For example, p_{34}^c becomes p_{34}.

Table 7.1. A sample Look Up Table (I^s = {1,3,{23},{34}}, ADR_{p_1}: address of p_1^c, $ADR_{p_{23}}$: address of p_{23}^c).

Communication Patterns	**Field I**	**Field II**
1	ADR_{p_1}	0
2	ADR_{p_3}	$ADR_{p_{23}}$
3	ADR_{p_3}	0
4	ADR_{p_3}	$ADR_{p_{34}}$

7.4.3. Extractor

In a communication instance, in the look up table, if entry II is null, then the partial bitstream is automatically downloaded to the FPGA after the decompression process. Otherwise an XOR operation is realized after the decompression between fields I and II as indicated in Equation 7.2. For example, for the 4th communication pattern we have to obtain the p_4, 4th row in Table 7.1. p_4 is extracted by applying an XOR operation to p_2 and p_{24}: $p_4 = p_2 \oplus p_{24}$.

The performance comparison of PCAP, cPCAP and c^{2}PCAP core engines is given in Section 11.3 of Chapter 11. There, the choice of smallest partial bitstream sets is summarized in Table 11.3.

8. MAPPING, ROUTING PROBLEMS FOR NOC AND RECONFIGURABLE INTERCONNECTS

In the last decade, IC manufacturers have been trying to find new ways of pushing limits of the performance. Since it was getting more and more difficult to make single core clock frequencies higher, technology trend started to shift using multi-core architectures at the beginning of 2000s. For the time being, reputable processor manufacturers such as Intel, AMD have offered to their customers 4-cores, 6-cores single chip processors. Even more, they offer two or more single chip multi-core processors to consumers. In addition to these, many new applications are multi-threaded and give better performances on multi-core architectures compared to their single-core counterparts. Moreover, parallelism is everywhere; the most of computation intensive applications are running on embedded multi-core architectures. However, when the application cannot be parallelized enough, or the application is not parallel by it's nature, it may give worse performance on a multi-core architecture. On a multi-core architecture, there are several smaller cores which run mostly lower frequencies. Hence, assigning a single-thread application to a multi-core architecture gives not good performance as on a high-clocked single core architecture. In addition to these, increasing the number of cores on multi-core architecture leads to an increase in the number of messages communicated between them. This may end up with a reduced performance and increased energy consumption. For that reason, the way of mapping application nodes on a regular or custom multi-core architecture plays an important role in the system performance. As well as the mapping algorithm, the routing algorithm has a considerable impact on the performance of communication network between cores.

Most of the applications on multi-core SoCs have non-uniform communication traffic patterns and they can be predicted statically [1]. In addition to this, most of the multi-core SoC applications do not have many different communication flows (number of edges in task graphs) and each of these cores mostly communicates with a few of other cores. Usually, the traffic flow of these applications is already known

beforehand [17].

As already mentioned previously (see Chapter 1), NoCs propose an encouraging solution for the communication problem in multi-core embedded systems. However, there are several issues, which affect the performance of NoC designs in terms of speed, area and power consumption. Hence, NoC designers must spent their times to solve these issues before their designs. The most important of these parameters can be summarized as follows:

- network topology
- mapping algorithm
- routing algorithm
- switching method
- router architecture
- link bandwidth

The most important of these parameters is the network topology [52]. There are several network topologies for NoCs. Some of them are 2-D mesh, torus, octagon, fat tree, butterfly and etc. As a result, the network topology, mapping of cores onto the target architecture and also routing have significant impact on the overall system performance. Hence, we focus on mapping and routing algorithms on regular 2-D mesh NoC architectures, which are mostly preferred in multi-core embedded SoCs.

In this part of the thesis, three different approaches for task to core mapping, routing for multi core architectures are examined. Here, we mostly focus on the network architectures such as NoC and reconfigurable interconnects for embedded systems.

In the first chapter of this part, we propose a mapping algorithm called Particle Filter Mapping (PFMAP) [7]; PFMAP is able to map task nodes onto the cores of tile-based NoC architectures such as regular, irregular and custom 2-D or 3-D topologies. PFMAP is inspired from systematic resampling algorithm for particle filters, in which all particles can run parallel and independently from each other. Based upon

the experimental results from applying PFMAP for various real life and synthetic applications onto the different topologies and architectures, the performance of the 2-D mesh architectures in terms of communication cost increased up to by 51% for irregular topologies, up to by 31% for custom architectures. Similarly, total travel distance obtained by PFMAP is reduced up to by 45% for custom 2-D Mesh architectures. In addition to these, average clock cycles per flit and total network power are reduced by up to 17% and 15% for regular 2-D mesh architectures respectively. Finally, communication cost is diminished by up to 34% for 3-D regular NoC architectures.

Afterwards, we propose a routing algorithm based on particle filtering algorithm. We call this approach as PFROUT (Particle Filter Routing). PFROUT is developed on the basis of PFMAP, it extends the mapping capabilities of PFMAP by also applying routing at the time of mapping. Likewise PFMAP, PFROUT also exploits systematic resampling algorithm for particle filters. In addition to these, PFROUT uses wavefront algorithm, and Dijkstra's shortest path in order to find a better routing for a given application on a network.

To deal with the communication bottleneck of multiprocessor systems, several communication architectures have been proposed in the last decade. Yet, none of them has demonstrated the performance of the direct connections between two communicating units as no additional components are required for mastering the communication. In Chapter 11, we propose dynamically reconfigurable point-to-point (DRP2P) interconnects for setting up direct connection between two communicating units before the communication starts. While PFMAP and PFROUT are developed for general NoC communication architectures, DRP2P is developed for custom architectures. Our most efficient on-chip self-reconfiguration core, c^2PCAP core is also used for the run-time reconfiguration of DRP2P interconnects.

8.1. Related Works on NoC Mapping Problem

A vast amount of methods have been proposed to solve mapping problem. In PMAP [53], two-phase mapping algorithm for placing clusters onto processors are used.

In NMAP [54], Dijkstra's shortest path on quadrant graph is applied to solve mapping problem. Both NMAP and PMAP are fast heuristic methods based on the approach of placing the most communicating nodes neighbour to each other by mapping heavy weight nodes at first. GMAP [55] is a greedy algorithm which uses n-ary search tree. The final configurations are given in the leaf nodes of their search tree. Although this algorithm uses branch-and-bound, space complexity of this work is in the factorial range. SUNMAP [56] extends NMAP to support new NoC topologies such as torus, hypercube. In CGMAP [57] a genetic algorithm using chaotic systems is presented. In Onyx [58] and Crinkle [59], priority lists are utilized. In [60], an Optimized Simulated Annealing (OSA) approach is proposed and tested on various task graphs. Two different algorithms, named A3MAP-GA and A3MAP-SR, have been presented in A3MAP [5,6]. A3MAP-GA is a genetic algorithm, while A3MAP-SR is a successive relaxation algorithm. There is an intensive survey on application mapping strategies for NoC design [61]. In this work, besides giving classification of mapping algorithms, communication cost of some benchmark applications are compared for various known algorithms. There are also studies trying to find optimum solution of the mapping problem by using Integer Linear Programming (ILP) [62]. Since the mapping problem is intractable, it is not possible to find a solution for medium and large size problems using ILP. In LMAP [63], Kernighan-Lin based partitioning is used to solve the mapping problem for regular architectures. In PSMAP [64], authors propose particle swarm optimization. Similar to [63], the scalability of this algorithm is also ambiguous, since it gives only a few sample applications mapped on a 2-D regular mesh architecture.

In most of these previous studies, video applications such as Video Object Plane Decoder (VOPD), MPEG4, and high-end video applications such as Picture in Picture (PIP), Multi Window Application (MWA), MWA with Graphics (MWAG), Dual Screen Display (DSD), Multi-Media System (MMS) including H263 Dec. and Enc., MP3 Dec. and Enc. are used. In addition to them, Embedded System Synthesis Benchmarks Suite (E3S) [65] and for synthetic task graphs Task Graphs for Free (TGFF) [2] are used in most of these studies.

Most studies are heuristic due to complexity. In most of these algorithms, for a

fixed problem size, the running time of the algorithms are fixed. However, in PFMAP, running time of the algorithm depends on the user. User can set a desired time to finish the algorithm. In PFMAP, as the number of samples and re-sampling iterations to generate new samples become larger, the solution approaches the optimal Bayesian estimate. In addition, in PFMAP, particles representing a configuration can run fully in parallel. Thus, PFMAP is very suitable for current and next generation multi-threaded or real parallel platforms such as multi-core, GPU and VPU.

8.2. Related Works for NoC Routing Problem

Traditionally, two types of NoCs have been proposed in the literature: packet-switched and circuit-switched. The data transfer time is much shorter in circuit-switched NoCs than it is in packet-switched NoCs, because a dedicated path between two nodes is established before communication takes place. However, the circuit set-up time introduces additional latency on the overall communication. Besides, the dedicated communication path might make other nodes wait for a communication channel. The packet-switched NoC solves this problem by limiting the size of data traveling in the NoC to the size of a packet. Yet, this type of NoCs suffer from not only process time for packetization of data, header processing or buffering but also communication time overhead due to congestion control [12].

Recently, methods that by-pass some of the routers are introduced so as to reduce the communication latency due to routers. In [1], authors propose a hybrid architecture, where they use simple switch boxes bypassing routers all the way. As the configuration switches are simpler than routers, they try to route communication paths through switches by minimizing the number of routers used. In PNoC [12] and DyNoC [13], modules are placed and removed dynamically. Here unused routers are reused for computing purposes. Topology switches are first introduced in ReNoC [14] where hybrid topologies can be setup by taking application specifications into account. In [15, 17,18], it is shown that by-pass links can also be introduced dynamically to the system. Similarly, in Reconfig-Net [16], the wires to and from the routers are dynamically added or removed. The number of routers is reduced by static analysis. In [66] these wires

Table 8.1. Comparison of latest NoC approaches.

Network	Switching	Device	Topology	Routing Alg.	Goal	Novelty	Application
AppAw [1]	Packet + Circuit	ASIC	2D-Mesh	Wormhole switching	Bypass the routers, use configuration switches	Use simple switch boxes between routers	VOPD, MWD, MP3 enc./dec., H.263 enc./dec.
PNoC [12]	Circuit	FPGA	Custom	Det.	Reduce Nr. of Routers compared to regular NoCs	Place modules that communicate frequently in the same subnet.	Image Binarisation
DyNoC [13]	Packet	FPGA	2D-Mesh	Adapted XY (S-XY, SV-XY, SH-XY)	Direct comm. paths btw. neighbor PEs	Use routers as reusable elements, i.e. also for logic	Color Generator and Traffic Light Controller
ReNoC [14]	Packet + Circuit	ASIC	Custom	Topology Switch+ Router	Reduce Nr. of Routers	Wrapping routers with topology switches	VOPD
Skip-links [15]	Packet	-	2D-Mesh	Adaptive	Jumping or skipping over the intermediate router	Connecting skip links together: skip chains.	-
Reconfig-Net [16]	-	FPGA	Custom	-	Reduce Nr. of Routers	Reconfigurable interconnects btw. routers	VOPD
VIP [17]	Packet	ASIC	2D-Mesh	Wormhole switching	Bypass the router pipeline stages with VCs	Use dedicated P2P links.	VOPD, MWD,H.263 enc., GSM, MP3 enc./dec.
RecoNoC [18]	Packet + P2P Simplex Links	FPGA	2D-Mesh	Adaptive Wormhole	Bypass routers using shortcuts	Create shortcuts btw. distant nodes	-
[66]	Packet	FPGA	2D-Mesh	Any deadlock-free alg.	Reducing the average distance between remote nodes	Introducing long-range links	Auto industry, Telecom and synthetic
[67]	Packet	FPGA	2D-Mesh	Wormhole	Using different architectures for data communication and synchronization	Hybrid architecture for multicast, broadcast and larger messages	Jacobi Algorithm
[68]	Packet + Circuit	Sim. Env.	2D-Mesh	Wormhole	Avoid router latency	Proposing virtual point to point links	SPLASH-2 applications [69]
[70]	Packet + Circuit	Sim. Env.	2D-Mesh	XY-Routing	Exploit advantages of both packet and circuit switching	Dynamically switching between packet and circuit switching	SPLASH-2 applications [69]

are static. In [67], three different communication architectures are utilized on the same SoC and one of these architectures is selected during run-time. This is accomplished by a switches and an adaptive look-up table at each router.

In addition to the abovementioned studies, there are newly proposed works on NoC architectures. In [68], authors propose a NoC architecture, where both packet switching and circuit switching combined in way that reduces overall packet latency. They propose virtual point-to-point (VIP) links, which are designed on the top of a packet switched network architecture. In order to avoid router's latencies they try to use VIPs all the way. If this is not possible they switch to the packet switched network. Similarly in [70], authors propose again a hybrid network topology, where packets starts to use at first packet switched network, and continues with circuit switched network, whenever such a circuit can be established. All above mentioned NoC approaches are summarized in Table 8.1.

As routers have a huge impact on the NoC performance and in order to reduce congestion delays, new router architectures are proposed recently [71,72]. As travelling packets through NoC are control and data packets, authors in [73], distinguish these packets by introducing a new router architecture. authors propose that most of the packets are the control packets and they can be sent to the destination via virtual channels.

Routers have a major effect on the occupied area and system power consumption. To avoid the drawbacks of routers, it is valuable to implement a routerless NoC-like system. From Table 8.1, it is obvious that almost all studies try to suppress the drawbacks of routers used in both packet and circuit switched networks. To achieve this, either the number of routers is reduced or routing through routers is minimized. This is done by either developing simpler router architectures or adding additional configuration switches or using dedicated P2P links for neighbour/long distant PEs and so on. Hence, PFROUT tries to minimize the congestion and contention delays for a target NoC architecture by minimizing the number of routers used on the communication architecture. Target architecture for PFROUT has additional simple configuration

switches in addition to the routers which are available on a conventional 2-D mesh NoC architecture. The main objective of PFROUT algorithm is routing communication requests through these simple switches instead of through power and area hungry, huge routers.

8.3. Related Works for Reconfigurable Interconnects

As traditional NoCs suffer from communication latency, area overhead and power consumption introduced to the system by the routers [17], adaptive, programmable, reconfigurable NoC architectures have been proposed in the literature. Adaptive look-up tables are proposed in PNoC [12]. In RecoNoC [18], parameterized look-up tables are used to reduce reconfiguration time. In PNoC, modules are placed and removed dynamically as it is the case in DyNoC [13] where the unused routers are reused for computing purposes. Some of these studies remain at simulation level [15] , but there exist implementations either as an ASIC [14] or on a Xilinx FPGA [12,13,16,18,66,67], or a Xilinx FPGA-based platform [17].

Xilinx FPGAs are well-known for the dynamic partial reconfiguration availability for almost every unit in the FPGA [10]. However, very few of the NoC implementations on the FPGAs utilize this property [12, 13, 16]. In these studies, power consumption and latency due to reconfiguration have not been reported. It is also ambiguous how reconfiguration is triggered and carried out in these studies. In the literature, there are agents and methods for dynamic partial reconfiguration, but these studies do not explain which one is utilized. Other run-time adaptive approaches propose new reconfiguration or adaptation methods based on run-time monitoring of the application. However, they do not explicitly mention how long their proposed reconfiguration/adaptation/monitoring scheme will take or how much power it will consume. The numerical time values for the proposed reconfiguration methods are not very promising. For example, TMAP [74] method adopted in RecoNoC requires 215 msec to update 8-bit coefficients of a 32-tap FIR. This is a quite long time for setting up communication architecture.

In DRP2P, we reconfigure interconnects between communicating modules for a given embedded application. The aim in DRP2P interconnects is setting up direct connections between two communicating units before the communication starts. We do not use any router for DRP2P. Communication between units is always established through direct connections. Hence, there is no router drawback for DRP2P architecture. DRP2P interconnects are examined in Chapter 11 in detail.

9. PARTICLE FILTERING ALGORITHM FOR NoC MAPPING PROBLEM

As NoC introduces scalable solution for multi-core architectures, it is a promising solution for the communication in a multi-core architecture. The first and most important parameter which affects the performance of NoC communication architecture is the mapping and placement of task nodes onto that architecture. For that reason, the way of mapping application nodes on a regular, irregular or custom multi-core architectures plays an important role in the system performance [75].

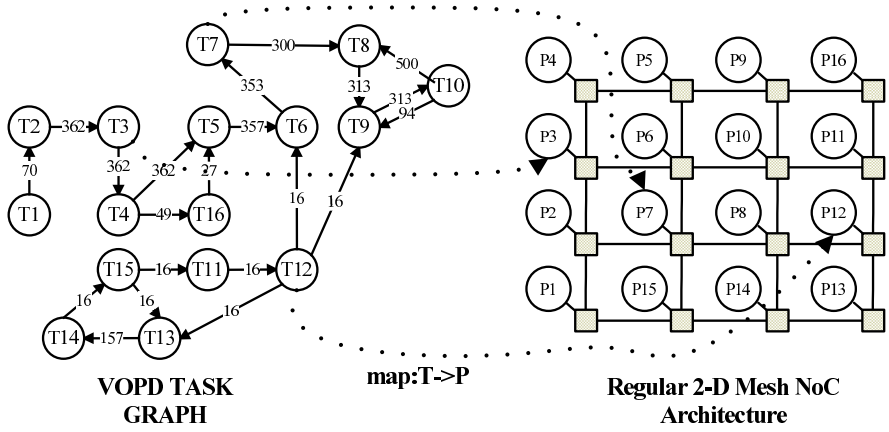


Figure 9.1. Task mapping process on a regular 2-D Mesh NoC.

Mapping process can be defined as assigning each task node to an individual core. Mapping can be divided into two subcategories as static and dynamic with respect to implementation time. Static mapping is done at compile time, whereas dynamic mapping runs on the fly and requires more complex components such as observer and reconfiguration engine. Moreover, dynamic mapping also requires an initial mapping. Our objective is the static mapping of cores on a regular, irregular and custom 2-D or 3-D mesh architectures by minimizing the communication cost. Static mapping is the first step to minimize overall packet latency and network power [5, 6, 54]. Hence, the keyword *mapping* refers to *static mapping* in the rest of the thesis.

A sample mapping can be found in Figure 9.1. On the left side of this figure, Video Object Plane Decoder (VOPD) application task graph [4] is given. Each node

in this graph represents a task (T) to be mapped onto a core (C).

A good mapping algorithm may also reduce the average traffic load on routers by placing most communicating nodes neighbor to each other as much as possible. Reducing the traffic load of routers may decrease the total area and power consumption of the system while increasing the operating system frequency (e.g. shortening the critical path).

The global objective of our PFMAP algorithm is the mapping of cores on a regular, irregular and custom 2-D or 3-D mesh architecture by minimizing the communication cost.

In the literature, there are various studies in the field of node to core mapping of regular NoCs. However, none of these studies utilize particle filtering algorithm to solve the mapping problem for NoCs, which is an NP-hard problem [61]. Yet, particle filters are widely used in applications such as positioning, localization, tracking and navigation in robotics, automotive industry [76–78]. Particle Filtering is a sequential Monte Carlo technique for the solution of the state estimation problem [79]. The original particle filtering algorithm is called Sequential Importance Resampling (SIR) and used frequently [80]. The main point in particle filtering is representing the required posterior density function (pdf) by a set of random sample particles with corresponding weights, and to compute the estimates based on these samples and weights. As the number of samples and resampling iterations to generate new samples become very large, the solution approaches the optimal Bayesian estimate. There are various resampling methods for particle filtering algorithm [81]. Systematic resampling algorithm [82] is widely used because it is easy to implement and it outperforms other resampling approaches in most scenarios. Moreover, in terms of resampling quality, systematic resampling has the minimal variance [83]. Hence, we have preferred to use systematic resampling in PFMAP.

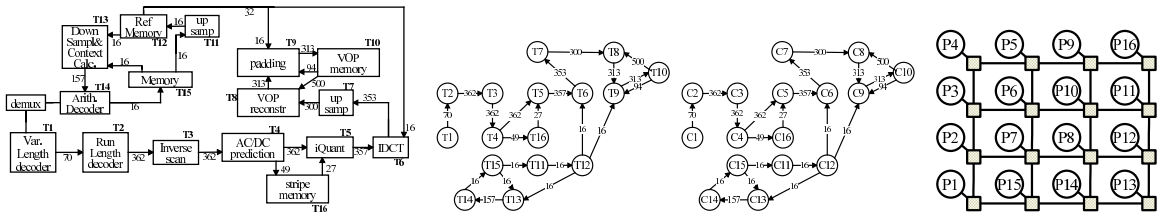
PFMAP is very suitable for solving mapping problem for the following reasons:

- A configuration on any type of regular, irregular and custom 2-D or 3-D NoC topology can be represented by particles.
- Particles do not require a fixed computation time; instead, accuracy increases with the available computational resources [77].
- Implementation of particle filters is extremely easy, especially systematic resampling algorithm.
- Particles give much better results than their counterparts in the solution of mapping problem in most of the time.
- Particles in PFMAP are totally independent from each other and therefore they all can run in parallel. Hence, it is easy to implement PFMAP on parallel computational platforms such as multi-thread, GPU and VPU.

9.1. Proposed Algorithm

Two graphs are the inputs to PFMAP. The first one is Task Traffic Graph (TTG), where the task nodes and the communication flows between them are defined. The second graph is Core Traffic Graph (CTG) in which processor or computational cores and their communication relationships are given. Another input is the topology of the NoC which is called Router Configuration Topology (RCT). We define configuration as the placement of cores on tile-based NoC architecture. As an example, the block diagram of VOPD application is given in Figure 9.2a. Each block in this application can be considered as a task node. In Figure 9.2b, task graph of VOPD application is given. This task graph is generated according to the block diagram in Figure 9.2a. Here, weighted directed edges represent the average communication volume in MBytes/s from one node to another node. The task graphs used in this thesis characterize the partitioning, task assignment, scheduling, communication patterns, and task execution time of a given application [55]. Similarly, Figure 9.2c shows the core graph of target application. In Figure 9.2d, a solution is found for mapping problem of VOPD application to a 2-D regular mesh NoC architecture by using NMAP algorithm. Here, rectangular shapes represent routers, circular shapes represent processor cores attached to routers.

Mathematical formulation of the mapping problem can be given as follows:



(a) Block Diagram (b) Task Traffic Graph (c) Core Traffic Graph (d) Mapping with NMAP [54]

Figure 9.2. VOPD application with 16-cores [4].

Definition 9.1. Task Traffic Graph (TTG) *is a directed graph, TTG(N,T) with each vertex* $n_i \in N$ *representing a task node and the directed edge between* n_i *and* n_j *indicated by* $f_{i,j} \in T, i \neq j$ *represents the traffic flow. The weight of* $f_{i,j} \in T$ *is the traffic amount from* n_i *to* n_j *and denoted by* $t_{i,j}$*. In TTG,* $|N|$ *represents the number of task nodes, while* $|T|$ *is the number of non-zero directed edges between* n_i *and* n_j*, s.t.* $i \neq j$*.* $|T| \leq |N| \times |N-1|$*.*

Definition 9.2. Core Traffic Graph (CTG) *is a directed graph, CTG(C,T) with each vertex* $c_i \in C$ *representing a processor or a computational core.The directed edge between* c_i *and* c_j *indicated by* $l_{i,j} \in T$ *represents the link between source and destination nodes. The weight of each link* $l_{i,j} \in T$ *is denoted by* $t_{i,j}$ *representing the traffic amount on the current link.*

One-to-one mapping of the TTG onto the CTG can be defined in Equation 9.1.

$$map : N \mapsto C, \ni map(n_i) = c_i, \forall n_i \in N, \exists c_i \in C \tag{9.1}$$

We assume that, each single task node $n_i \in N$ is mapped onto a single processor or computational core $c_i \in C$, on which there is no any other task node $n_j \in N$ is mapped yet. Hence, TTG and CTG are identical, i.e. $|N| = |C|$. In some applications, however, more than one task node can be mapped to a processor core or in the similar way, one task can be partitioned into subtasks and these subtasks can be mapped onto

multiple processor cores. We assume that these operations are carried out prior to PFMAP.

Definition 9.3. Router Configuration Topology (RCT) *is a 2-D mesh NoC topology with each ordered pair $P_{r,c} \in RCT(R, C)$ representing the physical location of a processor core attached to a router on the target architecture. R and C indicate the number of rows and columns in the topology respectively (i.e. $R \times C$ is NoC size). In $P_{r,c}$, r and c are the respective horizontal and vertical indices.*

One-to-one mapping of the CTG onto the RCT can be defined in Equation 9.2.

$$\begin{aligned} map : C \mapsto RCT, \ni map(c_i) = P_{r,c}, \\ \forall c_i \in C, \exists P_{r,c} \in RCT(R, C) \end{aligned} \tag{9.2}$$

Note that the mapping is valid if the number of cores to be placed ($|N|$) is less than or equal to the number of nodes on the target architecture ($|R \times C|$), $|N| \leq |R \times C|$.

RCT is composed of only routers and there are some limitations for routers in terms of their number of inputs and outputs: they have only n-bit single input and single output in one side (North-East-South-West). Each processor or computational core is attached to a router. Apart from the processor interface, both the maximum number of inputs and outputs for a router is four.

Definition 9.4. Manhattan Distance (MDist) *is the minimum number of hops from source node n_i to destination node n_j in RCT. The formula of the MDist for the nodes $P_{r1,c1}$ and $P_{r2,c2}$ in RCT is given in Equation 9.3.*

$$MDist = |r1 - r2| + |c1 - c2| \tag{9.3}$$

The communication cost of a configuration for regular 2-D mesh architectures is calculated by using $MDist$ between each node pairs. Communication cost for a single edge (CC_{se})between $P_{r1,c1}$ and $P_{r2,c2}$ in a configuration is given in as,

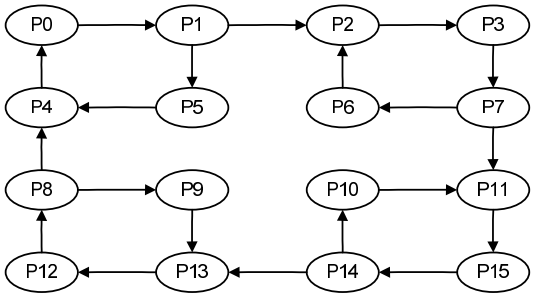
$$CC_{se} = t_{P_{r1,c1},P_{r2,c2}} * MDist(r1, c1, r2, c2) \tag{9.4}$$

$CommCostReg$ is the total communication cost of a configuration for a regular 2-D mesh topology as shown in Equation 9.5. Here, R and C indicate number of rows and columns in the RCT respectively.

$$CommCostReg = \sum_{r1=0}^{R} \sum_{c1=0}^{C} \sum_{r2=0}^{R} \sum_{c2=0}^{C} CC_{se}(r1, c1, r2, c2) \tag{9.5}$$

Instead of using $MDist$, distance value of each computation core pairs for irregular, custom and 3-D architectures is obtained by utilizing the Dijkstra's shortest path algorithm [84] in the preprocessing step, where a distance matrix is generated. Then, this matrix is used as an input to PFMAP. In Figure 9.3, a random processor communication topology and its corresponding distance matrix are given. Source and destination processor cores are given in the rows and columns of the distance matrix. For example, the distance from source node $P6$ (in row $P6$) to destination node $P5$ (in column $P5$) is given as thirteen. The path from $P6$ to $P5$ is as follows: $P6 -> P2 -> P3 -> P7 -> P11 -> P15 -> P14 -> P13 -> P12 -> P8 -> P4 -> P0 -> P1 -> P5$.

In Equation 9.6, $CommCostIrreg$ is the total communication cost of a configuration for an irregular, custom 2-D mesh or regular, irregular and custom 3-D mesh topologies.



	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
P0	0	1	2	3	3	2	5	4	10	11	8	5	9	8	7	6
P1	3	0	1	2	2	1	4	3	9	10	7	4	8	7	6	5
P2	10	11	0	1	9	12	3	2	8	9	6	3	7	6	5	4
P3	9	10	3	0	8	11	2	1	7	8	5	2	6	5	4	3
P4	1	2	3	4	0	3	6	5	11	12	9	6	10	9	8	7
P5	2	3	4	5	1	0	7	6	12	13	10	7	11	10	9	8
P6	11	12	1	2	10	13	0	3	9	10	7	4	8	7	6	5
P7	8	9	2	3	7	10	1	0	6	7	4	1	5	4	3	2
P8	2	3	4	5	1	4	7	6	0	1	10	7	3	2	9	8
P9	5	6	7	8	4	7	10	9	3	0	13	10	2	1	12	11
P10	8	9	10	11	7	10	13	12	6	7	0	1	5	4	3	2
P11	7	8	9	10	6	9	12	11	5	6	3	0	4	3	2	1
P12	3	4	5	6	2	5	8	7	1	2	11	8	0	3	10	9
P13	4	5	6	7	3	6	9	8	2	3	12	9	1	0	11	10
P14	5	6	7	8	4	7	10	9	3	4	1	2	2	1	0	3
P15	6	7	8	9	5	8	11	10	4	5	2	3	3	2	1	0

(a) An irregular processor topology (b) Distance matrix

Figure 9.3. An irregular processor communication topology and it's distance matrix.

$$CommCostIrreg = \sum_{i=0}^{E} \sum_{j=0}^{E} t_{i,j} * dist_{i,j} \tag{9.6}$$

where $dist_{i,j}$ is the Dijkstra's shortest path.

Definition 9.5. CostLowerBound *is the minimum communication cost that can be achieved in a given configuration. CostLowerBound calculation is given in Equation 9.7.*

$$CostLowerBound = \sum_{i=0}^{E} \sum_{j=0}^{E} t_{i,j} \tag{9.7}$$

The main part of our PFMAP mapping algorithm is given in Figure 9.4. In this algorithm each particle represents a mapping configuration. The algorithm generates PN random configurations in the first iteration (lines 4-6). The related function ($randomConf(particles_j)$) is an implementation of Fisher-Yates Shuffle [85], which is used for the sake of performance. With this function, we randomly place the processor or computational cores on mesh NoC architecture initially for each configuration. Here, PN is the number of particles and IT is the number of iterations, that we define before the beginning of running our algorithm. It should be noted that the running time of

our algorithm is proportional to the PN and IT.

Input: Set of cores and nodes, TTG, CTG, RCT(R,C)

Output: Mapping of cores to nodes

```
1: BestFitness = −1
2: for i = 0 to IT do
3:    for j = 0 to PN do
4:       if (i == 0) then
5:          randomConf(particles_j) ;
6:       else
7:          randomNodeSwap(particles_j) ;
8:       end if
9:       particleFitness_j ⇐ CalcFitness(particles_j) ;
10:      CurrFitness ⇐ particleFitness_j ;
11:      if CurrFitness > BestFitness then
12:         BestFitness ⇐ CurrFitness
13:         BestConf ⇐ particles_j
14:         BestCost ⇐ CommCost for particles_j
15:      end if
16:   end for
17:   sysResamp(particleFitnesses, chosens, PN)
18:   chooseResampleds(particles, chosens, PN)
19: end for
```

Figure 9.4. Main part of configuration mapping algorithm.

In the first iteration, we calculate the fitness for each randomly generated configuration (lines 9-10). The calculation of fitness function for mapping is given in Equation 9.8. Here, $CommCost$ is identical to $CommCostReg$ if the architecture is regular 2-D mesh, otherwise (i.e. architecture is irregular, custom 2-D mesh or regular, irregular, custom 3-D mesh) it is equal to $CommCostIrreg$.

$$Fitness_{mapping} = 1/CommCost \tag{9.8}$$

According to initial configurations, we calculate the fitness value for each configuration. Among these configurations, the largest fitness value is set as the minimum $BestFitness$ (lines 11-15).

Input: Set of cores and nodes, TTG, CTG, RCT

Output: Mapping of cores to nodes

1: **Function** randomConf(RCT(R,C))
2: **for** $i = 0$ to N **do**
3: $tempArr_i \Leftarrow i$
4: **end for**
5: $numWaitingChips \Leftarrow N$
6: **for** $i = 0$ to ROW **do**
7: **for** $j = 0$ to COL **do**
8: $randInt \Leftarrow MSTrand(0, numWaitingChips)$
9: $RCT_{i,j} = tempArr_{randInt}$
10: $tempArr_{randInt} = tempArr_{--numWaitingChips}$
11: **end for**
12: **end for**
13: **RETURN** $modifiedRCT$
14: **EndFunction**

Figure 9.5. Random configuration function.

After generating initial configurations and finding the *BestFitness*, we re-sample these configurations in each iteration (lines 17-18). Sampling from the distribution and checking those samples with largest fitness values can be utilized for a *low cost configuration selection algorithm*. In the remaining IT-1 iterations, we apply pairwise swap among randomly selected nodes (line-7). At the initial steps, the algorithm might not represent the fitness function. However after a *burn-in* period, it starts to converge to the distribution. The *burn-in* period is directly proportional to the application and NoC architecture size.

In Figure 9.5, the generation of random configurations (i.e. particles) is given for the first iteration of the Figure 9.4. In this algorithm Fisher-Yates Shuffle [85] is used

for the sake of performance.

In Figure 9.6, swapping of distinct node pairs on RCT is given.

For all random function generations (lines 5 and 7), we used thread-safe SIMD-oriented Fast Mersenne Twister (MT) Pseudo Random Number Generator (PRNG) [86] because of the following reasons:

- It has larger period (up to $2^{216091} - 1$) than the original MT ($2^{19937} - 1$) [86].
- It is roughly twice faster than the original MT, and has a better equidistribution property, as well as a quicker recovery from zero-excess initial state [86].
- It is faster than other statistically reasonable generators (very useful when huge quantities of random numbers are required) [87].
- Original MT is proven to be equidistributed (up to 623-dimensional) for 32-bit values. It passes many stringent statistical tests, including the diehard test of G. Marsaglia and the load test of P. Hellekalek and S. Wegenkittl [88].
- It is very common; it has strong support from the people knowledgeable in the same field.

Input: RCT(R,C)
Output: RCT(R,C) with random two nodes swapped
1: **Function** randomNodeSwap (RCT(R,C))
2: do
3: {
4: pick first node from RCT randomly
5: pick second node from RCT randomly
6: }while($firstnode \neq secondnode$);
7: swap($firstnode, secondnode$) in RCT
8: save RCT as $modifiedRCT$
9: **RETURN** $modifiedRCT$
10: **EndFunction**

Figure 9.6. Random swap node pair function.

Input: Particle fitnesses and set of particles

Output: Set of re-sampled particles

1: **Function** sysResamp(particleFitnesses, particles)
2: $fitnessSum = 0$
3: $curFitnessSum = 0$
4: $uni = 0$
5: **for** $i = 0$ to P **do**
6: $\quad fitnessSum+ = particleFitnesses_i$
7: **end for**
8: $uni = ([0..1] \times (fitnessSum/P))$
9: $k = -1$
10: **for** $j = 0$ to P **do**
11: $\quad$ **while** $curFitnessSum < uni$ and $k < P - 1$ **do**
12: $\quad\quad k++$
13: $\quad\quad curFitnessSum+ = particleFitnesses_k$
14: $\quad$ **end while**
15: $\quad resampledParticles_j \leftarrow k$
16: $\quad uni+ = (fitnessSum/P)$
17: **end for**
18: **RETURN** $resampledParticles$
19: **EndFunction**

Figure 9.7. Re-sample particles systematically.

In Figure 9.7, method for re-sampling of particles is given. Here, configurations can be considered as a probability distribution where each configuration's probability is given by Equation 9.8 So, as the communication cost of a configuration increases, the probability of its selection decreases for the next iteration.

In systematic re-sampling, new configurations (i.e. particles) are derived from the previous ones. Number of configurations at the input and output are the same. While the configurations with high *CommCost* values are mostly discarded, each configuration with lower *CommCost* value is reproduced a few times. Thus, at the end of re-sampling, probably there will not be only a single instance of a configuration with low *CommCost*.

For re-sampling step, fitness of each configuration is calculated. Then, the fitness values are located in a one-dimensional array. The total size of this array is the sum of fitness values ($fitnessSum$ in line 6 of Figure 9.7) of P configurations. Here, the size of occupied area of each configuration is proportional to its size. Assume that we have three configurations $C1, C2, C3$ and they have *CommCost* values as $1, 0.2$, and 0.5 respectively. So, the corresponding fitness values are $1, 5$ and 2 respectively according to Equation 9.8. Residing of these configurations in the array is illustrated in Figure 9.8.

In our re-sampling scheme, we define a comb with P teeth, which selects (i.e. re-samples) the new configurations from the array. For our example, our comb has three teeth as in Figure 9.8. Teeth of comb select the appropriate configurations. Here, the length of comb is shorter than the length of array and the interval between teeth are equal. For P configurations, the length of comb can be given as:

$$CombLength = \frac{\sum_{i=1}^{|P|} p_{configuration_i}}{P} \times (P - 1) \tag{9.9}$$

For our example, the length of comb, $CombLength = [(1 + 5 + 2)/3] * 2 = 16/3$ and array length is eight. To locate the comb over the array, we generate a number from the

uniform distribution on the interval $[0, ArrayLength - CombLength]$. This number, uni, is the leftmost tooth of the comb (line 8). For our example, this interval is $[0, 8/3]$ and assume that uni is 1.32. The process of locating the comb over the array and selecting new candidates are done in lines 10-17 and shown in Figure 9.8. In lines 11-14, decision about the current configuration is given. If the sum of cumulative sum of fitness values ($curFitnessSum$) and fitness of the current configuration ($particleFitnesses_k$) is less than current value of uni (i.e. $k * (fitnessSum/P)$), k is incremented. Thus, the kth particle is not re-sampled; instead, $(k-1)$th particle is reproduced.

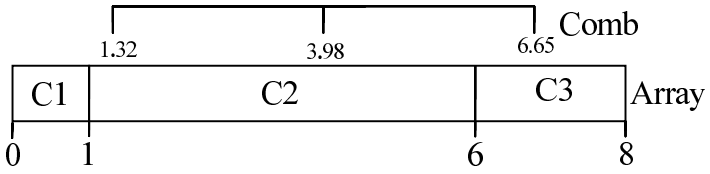


Figure 9.8. Residing of three configurations on an array for re-sampling step by using comb.

Our re-sampling method generates only a single real number. Hence, in the next iteration, the probability of having better configurations is increased while still keeping some of the configurations with higher costs as well.

9.2. Case Studies

We implement our PFMAP algorithm in C++ with OPENMP library [89]. All tests have been carried out on a 32-bit Windows-7 PC with a i5 CPU-750@2.67GHz and 3-GB RAM. We performed our experiments with various video applications such as VOPD, MPEG4-Decoder, MWD, MMS-Suite (H263-decoder, H263-Encoder, MP3-Decoder, MP3-encoder), E3S Benchmark Suite [65] (Automotive/ Industrial (AI), Consumer, Telecommunications). Additionally, we have generated various synthetic task graphs using TGFF [2]. These applications are mapped onto regular, irregular and custom 2-D, 3-D NoC architectures.

For the simulation purposes, we have used NIRGAM NoC simulator [90]. NIRGAM is a SystemC based cycle-accurate NoC simulator for 2-D regular NoC architectures. Yet, it does not support multi-casting. Moreover, user cannot give the mapping as

Table 9.1. Algorithm running time and communication cost results of PFMAP on different applications.

Nr. Of Tests= 100								
Application	**IT**	**10**	**10**	**100**	**100**	**1000**	**1000**	**10000**
	PN	**10**	**100**	**100**	**1000**	**1000**	**10000**	**10000**
H263 dec (V=12, E=14) LB: 213175 OP: 213372	**Run T. [ms]**	1.21	1.41	13.35	28.53	295	2036	21002
	Best C.	218378	213372	213372	213372	213372	213372	213372
	Avg C.	243168	219494	213785	213464	213444	213374	213372
	W. C.	295814	228920	216991	213660	213660	213660	213372
H263 enc (V=12, E=11) LB: 403817 OP: 404014	**Run T. [ms]**	1.19	1.42	13.14	-	-	-	-
	Best C.	406306	404014	404014	-	-	-	-
	Avg C.	523072	433431	404014	-	-	-	-
	W. C.	664716	522249	404014	-	-	-	-
MP3 dec (V=12, E=4) LB: 42420 OP: 42420	**Run T. [ms]**	1.32	1.55	-	-	-	-	-
	Best C.	42420	42420	-	-	-	-	-
	Avg C.	44197	42420	-	-	-	-	-
	W. C.	56550	42420	-	-	-	-	-
MP3 enc (V=12, E=8) LB: 70679 OP: 77740	**Run T. [ms]**	1.20	1.33	13.39	-	-	-	-
	Best C.	77740	77740	77740	-	-	-	-
	Avg C.	79557	77780	77740	-	-	-	-
	W. C.	100268	78380	77740	-	-	-	-
MWD (V=12, E=12) LB: 1120 OP: 1216	**Run T. [ms]**	1.25	1.37	13.59	31.12	316.22	1980	-
	Best C.	1408	1312	1216	1216	1216	1216	-
	Avg C.	1800	1558	1364	1264	1224	1216	-
	W. C.	2080	1792	1504	1344	1248	1216	-
MPEG4 Dec (V=12, E=21) LB: 3466 OP: 3633	**Run T. [ms]**	1.22	1.36	13.42	29.19	297	1995	-
	Best C.	4272.5	3752	3633	3633	3633	3633	-
	Avg C.	4900.7	4030.1	3693.3	3643.9	3637.6	3633	-
	W. C.	5275.5	4602	3772.5	3672	3672	3633	-
VOPD (V=16, E=21) LB: 3731 OP: 4119	**Run T. [ms]**	1.23	1.47	14.51	36.07	329	2516	25033
	Best C.	4950	4561	4167	4125	4119	4119	4119
	Avg C.	6608	5583	4670	4225	4136	4129	4124
	W. C.	7946	6321	5178	4469	4157	4135	4135

Table 9.2. Algorithm running time and communication cost results of various studies.

Application	**Algorithms**					
		NMAP	**LMAP**	**PSMAP**	**ILP**	**PFMAP**
MPEG4(4x4)	**Run T. [s]**	0.024	0.040	0.040	21.53	0.005
	Comm C.	3672	4006	3567	3567	3567
MWD(4x4)	**Run T. [s]**	0.016	0.030	0.020	200.5	0.015
	Comm C.	1184	1248	1120	1120	1120
VOPD(4x4)	**Run T. [s]**	0.024	0.040	0.260	21.53	0.329
	Comm C.	4265	4189	4119	4119	4119

an input to the system. Hence, we modified NIRGAM to support these features. In NIRGAM, we selected the simulation frequency as 1GHz and set simulation time to 1ms. All other settings are left at their default values. It is enough to use hop count ($CommCost$) in order to evaluate quality of a mapping in terms of consumed energy [91] for regular architectures. The average energy consumption of sending one bit of data from one node (t_i) to another one (t_j) is determined by the Manhattan Distance for regular architectures:

$$E_{bit}^{t_i,t_i} = n_{hops} \times E_{S_{bit}} + (n_{hops} - 1) \times E_{L_{bit}} \tag{9.10}$$

In Equation 9.10, n_{hops} is the number of routers the bit traverses from tile t_i to t_j. $E_{S_{bit}}$ and and $E_{L_{bit}}$ are the energy consumed by the switches and links between tiles, respectively. Since $E_{S_{bit}}$ and $E_{L_{bit}}$ are dependent on NoC architecture, n_{hops} determines the energy consumption for regular architectures and it is directly related to mapping process.

9.2.1. 2-D Regular Mesh Architectures

In Table 9.1, various applications are mapped with PFMAP onto regular 2-D Mesh architectures. The results show both running time of our PFMAP algorithm and

the communication cost of each scenario for different number of iterations (IT) and particles (PN). In Table 9.1 and in the following illustrations, while V represents the number of task nodes, E shows the number of edges. For each application, the lower bound communication cost (LB) is the $CostLowerBound$ in Equation 9.7. However, sometimes it is impossible to reside all communicating task nodes as neighbor to each other. Hence optimum solutions (OP) need not be equal to lower bound cost values.

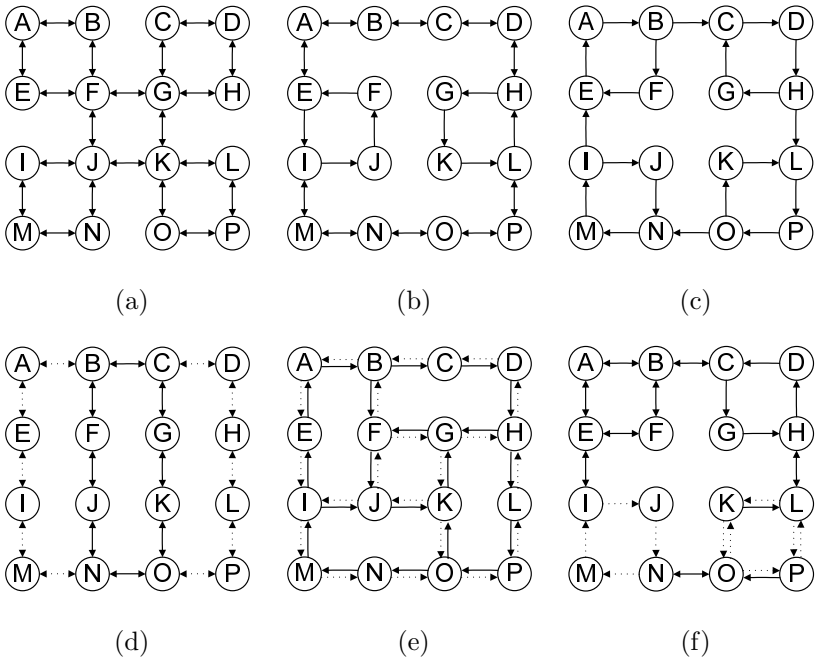


Figure 9.9. Irregular mesh architectures [5].

The halting methodology of our algorithm is as follows: if the resulting communication costs in the best case and worst case are close to the value of average case, we terminate the execution. In Table 9.1, running time ($Run\ T.$) of our PFMAP algorithm is given in milliseconds. The best ($Best\ C.$), worst ($W.\ C.$) and average case ($Avg.\ C.$) communication costs are also presented. We tested each benchmark 100 times for given IT and PN values. Generally, as IT and PN increase, we find better solutions. However, for some applications (i.e for small and simple applications) it does not make any sense to run it for large values of IT and PN. For example, for the H263 encoder application, we found the solution for $IT = 100$ and $PN = 100$ in 1.42 milliseconds. Hence, there is no need to run this application for larger IT and PN values anymore. For the medium size problems (e.g. VOPD with sixteen cores on 4x4 2-D mesh NoC), it is not easy to apply our halting methodology. For that reason, after some burn-in

period (if the resulting worst case solution is near to the best solution), we stop the running of our PFMAP algorithm. In such a situation, a designer can set a threshold value (e.g. 105% of the best communication cost) and check the average and worst values. If they are less than the threshold value, the algorithm might be halted.

We also compared the communication cost and running time of PFMAP algorithm with NMAP, LMAP, PSMAP and ILP studies in Table 9.2. In this table, ILP shows the optimum communication cost values and it is remarkable that PFMAP also finds optimum results for given applications in a short period of time. Although NMAP and LMAP are fast algorithms, they do not find optimum results for given medium size applications. In average, PFMAP seems to be the best algorithm among the other ones in terms of both communication cost and algorithm running time.

We have tested the performance of PFMAP on a fixed, 4x4 2-D regular mesh network with increasing communication demand between cores. To implement this, we have generated five task graphs with fixed number of vertices but different number of communication demands by using TGFF. In Figure 9.10, we see that PFMAP outperforms NMAP when communication demand increases.

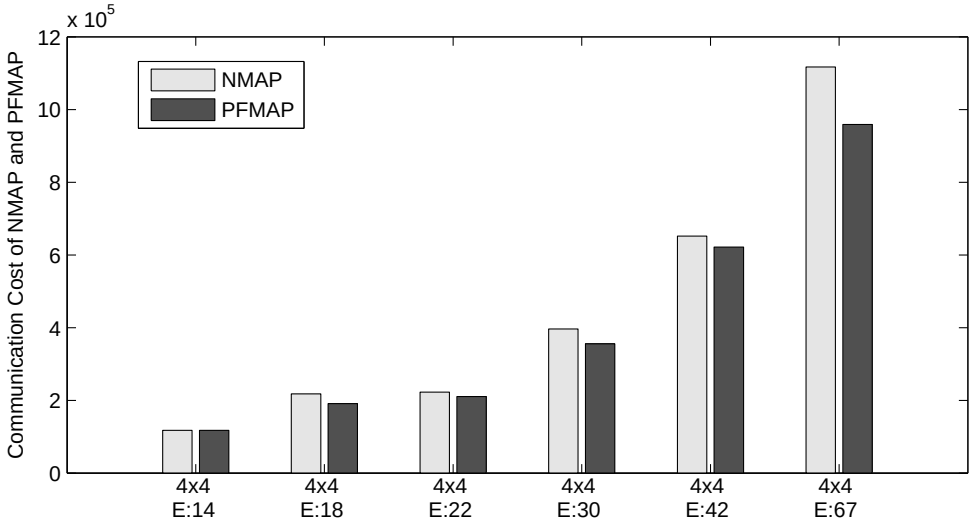


Figure 9.10. Communication cost comparison of PFMAP and NMAP on a 2-D NoC with fixed size (4x4) with increasing communication demand.

Running times of NMAP and PFMAP algorithms for the networks in Figure 9.10

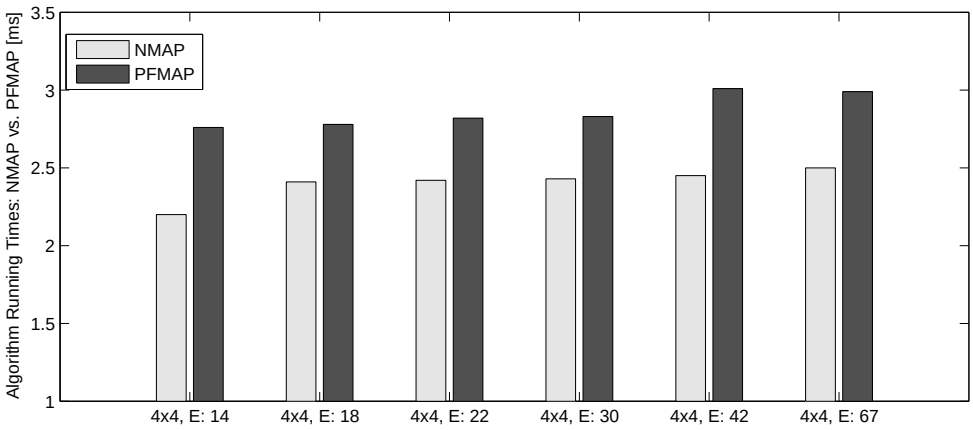


Figure 9.11. Algorithm running time of NMAP and PFMAP on a 2-D NoC with fixed size (4x4) with increasing communication demand (IT=10, PN=10 for PFMAP).

are given in Figure 9.11. Except the simplest one (i.e. graph with 14 edges), PFMAP finds a better result than NMAP with a little time overhead.

9.2.2. 2-D Irregular and Custom Mesh Architectures

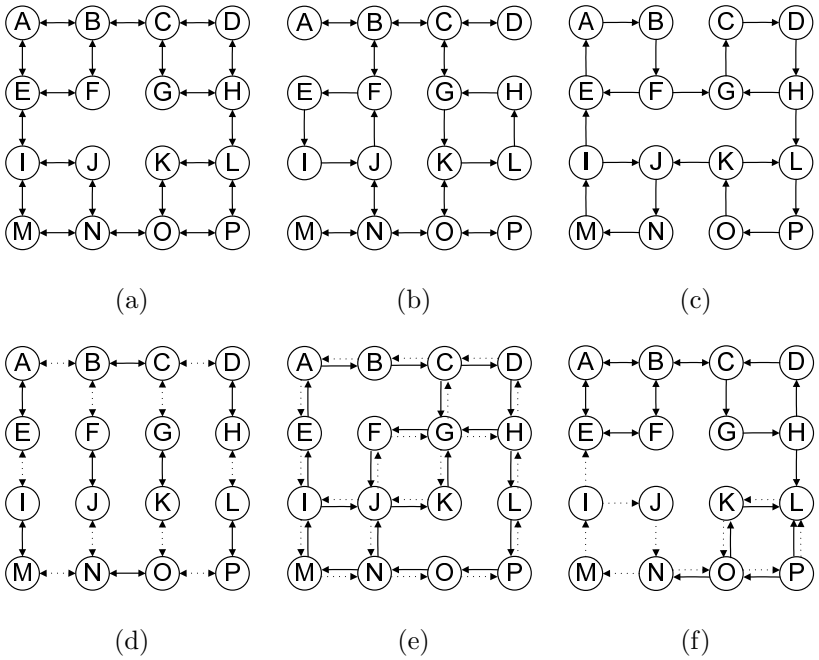


Figure 9.12. Irregular mesh architectures [6].

We have compared PFMAP with other studies such as NMAP, CMAP, A3MAP-SR [6] and A3MAP-GA [6] for VOPD application on some irregular 2-D mesh architectures given in Figure 9.12. Here, solid lines represent links with full bandwidth, while

dashed lines show the links width half bandwidth. PFMAP tries to place heavy weight communications onto the links with full bandwidth and the remaining smaller weight edges onto the links with half bandwidth irregular 2-D mesh topologies. As it is seen in Table 9.3, although PFMAP finds a worse communication cost in a few scenarios (see rows 7-8), it gives much better results than all it's counterparts in average for each scenario. Even though the PFMAP's resampling algorithm works very well, it might give good results only for large number of IT and PN values for a given application. We fixed both IT and PN values to 1000 for this set of experiments. If we increase these values, the PFMAP algorithm will probably find better results with the time.

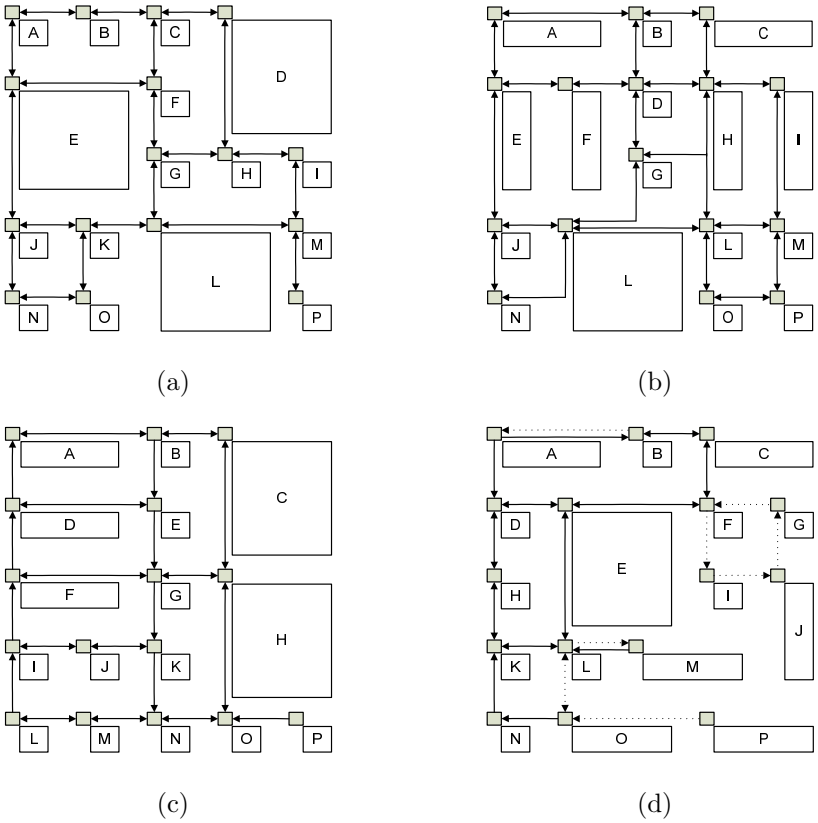


Figure 9.13. Custom mesh architectures [6].

We have also compared PFMAP with other studies for VOPD application on some custom architectures given in Figure 9.13. Comparison results are given in Tables 9.4 and 9.5. Communication cost of PFMAP is much better than other algorithms in average in all scenarios. Total travel distance of PFMAP might be worse than the other algorithms for a few scenarios (row 4 in Table 9.5) due to fixed IT and PN values.

Table 9.3. Communication cost of VOPD application on six different irregular mesh architectures (4x4).

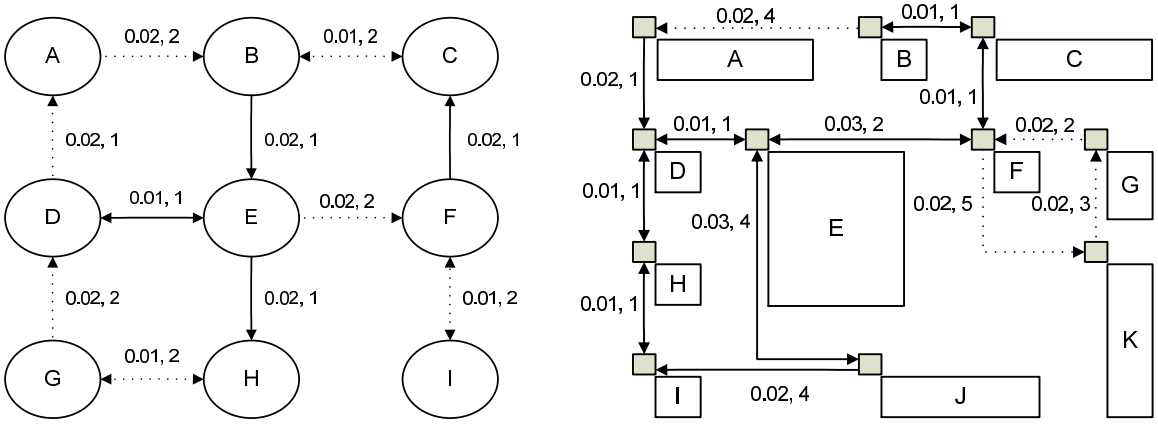
Total Communication Cost									
Network	**NMAP**	**CMAP**	**A3MAP-SR**	**A3MAP-GA**	**PFMAP**	**Imp. Over NMAP(%)**	**Imp. Over CMAP(%)**	**Imp.Over A3MAP-SR(%)**	**Imp.Over A3MAP-GA(%)**
Figure 9.12a	4215	4911	4205	4189	4189	+0.62	+14.70	+0.38	0.00
Figure 9.12b	8704	6544	5820	5345	4253	+51.14	+35.01	+26.92	+20.43
Figure 9.12c	6405	7194	6185	5257	4900	+23.50	+31.89	+20.78	+6.79
Figure 9.12d	4923	5507	4374	4199	4199	+14.71	+23.75	+4.00	0.00
Figure 9.12e	4950	4259	4191	4189	4281	+13.52	-0.52	-2.15	-2.20
Figure 9.12f	7424	6497	4832	4081	4351	+41.39	+33.03	+9.95	-6.62
Average	6104	5819	4935	4543	4362	+24.14	+22.98	+9.98	+3.07
Ratio	1	0.953	0.808	0.744	0.715				

Table 9.4. Communication Cost of VOPD application on four different custom mesh architectures (4x4).

Total communication cost									
Network	**NMAP**	**CMAP**	**A3MAP-SR**	**A3MAP-GA**	**PFMAP**	**Imp. Over NMAP(%)**	**Imp. Over CMAP(%)**	**Imp.Over A3MAP-SR(%)**	**Imp.Over A3MAP-GA(%)**
Figure 9.13a	4488	4752	4531	4087	4076	+9.18	+14.23	+10.04	+0.27
Figure 9.13b	4264	4119	4248	4199	3859	+9.49	+6.31	+9.16	+8.09
Figure 9.13c	6296	5598	5867	5150	4290	+31.86	+23.37	+26.88	+16.70
Figure 9.13d	5524	5735	4263	4263	4236	+23.32	+26.14	+0.63	+0.63
Average	5143	5051	4727	4425	4115	+18.46	+17.51	+11.68	+6.42
Ratio	1.000	0.982	0.919	0.860	0.800				

Table 9.5. Total travel distance (wirelength) by all packets.

Total travel distance									
Network	**NMAP**	**CMAP**	**A3MAP-SR**	**A3MAP-GA**	**PFMAP**	**Imp. Over NMAP(%)**	**Imp. Over CMAP(%)**	**Imp.Over A3MAP-SR(%)**	**Imp.Over A3MAP-GA(%)**
Figure 9.13a	5879	6300	5332	4543	4108	+30.12	+34.79	+22.96	+9.58
Figure 9.13b	5505	4135	5049	4215	4486	+18.51	-8.48	+11.15	-6.42
Figure 9.13c	7835	6842	7434	5613	4862	+37.95	+28.94	+34,60	+13.38
Figure 9.13d	9196	9627	5170	5170	4971	+45.94	+48.36	+3.85	+3.85
Average	7104	6726	5746	4885	4606	+35.16	+31.52	+19.84	+5.71
Ratio	1.000	0.947	0.809	0.688	0.648				



(a) Irregular mesh 3x3 (b) Custom with 11-cores

Figure 9.14. Energy, latency representation of irregular and custom architectures.

Hop count may not be sufficient to qualify the mapping quality of irregular and custom architectures [92]. Mapping quality also depends on the communication energy and latency for such architectures. To examine this issue, an irregular 3x3 mesh architecture and a custom 11-core architecture are given in Figure 9.14. Number pairs on the edges denote relative communication energy and latency of each link respectively. The communication latency from *Core i* to *Core j* is denoted by $l_{i,j}$ and obtained by the sum of relative communication latencies on the shortest path from *Core i* to *Core j*. The total communication latency (L_{Comm}) for an application mapping is given in Equation 9.11.

$$L_{Comm} = \sum_{i=0}^{E} \sum_{j=0}^{E} t_{i,j} * l_{i,j} \tag{9.11}$$

Similarly, the communication energy from *Core i* to *Core j* is denoted by $e_{i,j}$ and obtained by the sum of relative communication energies on the shortest path from *Core i* to *Core j*. The total communication energy (E_{Comm}) for an application mapping can be calculated as in Equation 9.12.

$$E_{Comm} = \sum_{i=0}^{E} \sum_{j=0}^{E} t_{i,j} * e_{i,j} \tag{9.12}$$

In this set of experiment, various benchmarks with different sizes generated by TGFF are mapped onto miscellaneous irregular and custom NoC architectures similar to Figure 9.14, but with different dimensions. Table 9.6 shows the communication energy and communication latency. Here, PFMAP gives always much better results than NMAP in terms of both communication latency and energy with a small running time overhead.

Table 9.6. Communication energy and latency comparison of NMAP and PFMAP on both irregular and custom architectures.

Network	**Benchmark**	**NMAP**			**PFMAP**					
					IT=100, PN=100			**IT=100, PN=1000**		
		L_{comm}	E_{comm}	*Run T.*$[ms]$	L_{comm}	E_{comm}	*Run T.*$[ms]$	L_{comm}	E_{comm}	*Run T.*$[ms]$
4x4 irregular	**TGFF16**	466.44	34626	2.63	395.72	28026	107	372.51	26170	173
Ratio		1	1	1	0.85	0.80	40	0.79	0.75	65
5x5 irregular	**TGFF25**	10438.77	838272	5.76	9173.19	715883	111	9085.77	673411	318
Ratio		1	1	1	0.87	0.85	19	0.87	0.80	55
6x6 irregular	**TGFF36**	156244.55	14980839	11.50	23336.87	1812894	149	21142.23	1582556	841
Ratio		1	1	1	0.14	0.12	13	0.13	0.10	73
11-core custom	**TGFF11**	337.83	44657	1.47	230.78	27747	84	225.23	26672	129
Ratio		1	1	1	0.68	0.62	57	0.66	0.59	87
17-core custom	**TGFF17**	15667.71	1675886	2.82	6232.33	665284	105	6038.62	656285	174
Ratio		1	1	1	0.39	0.39	37	0.38	0.39	61
28-core custom	**TGFF28**	18909.16	1992570	7.24	15934.52	1778346	128	14880.78	1508527	437
Ratio		1	1	1	0.84	0.89	17	0.78	0.75	60

9.2.3. 3-D NoCs

A 3-D NoC interconnection architecture is composed of 2-D layers connected to each other through vertical links. In Figure 9.15, a 3-D NoC architecture with dimensions X=4, Y=4 and Z=3 is given. Most attractive way to connect these layers is utilizing Through Silicon vias (TSV). However, TSV pads between layers occupy significant chip area and lead to congestion delays [93]. Hence, finding a good mapping algorithm, which decreases the number of TSVs, may increase the system performance

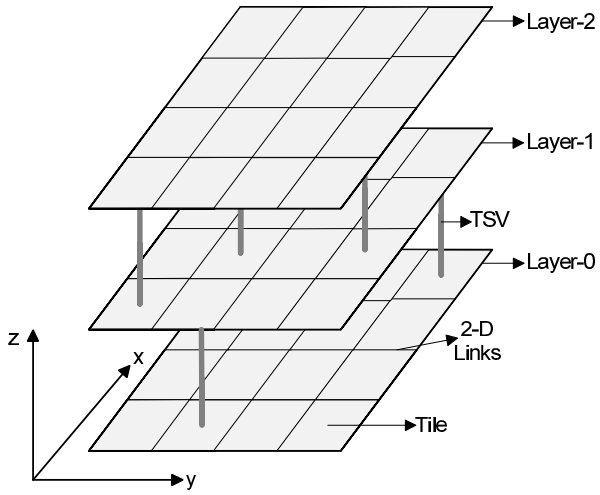


Figure 9.15. A 3-D NoC architecture with the size of 4x4x3.

by saving chip area, reducing communication delay. From this point of view, we applied PFMAP algorithm to applications such as AI, Telecom, DMC, MMS with 25-cores (MMS25) and MMS with 40-cores (MMS40) onto different size of 3-D NoCs.

Initially, we determine the 3-D NoC network dimensions according to the number of application task nodes. Given N as the number of task nodes for an application. The calculation of 3-D NOC dimensions is given in Figure 9.16.

After determining dimensions of target 3-D NoC, we assume all mutual tiles in neighbor layers are connected TSVs. Since TSVs are more costly than 2-D links, we set the cost of a TSV as five times of a 2-D link's cost. As the resampling iterations increase, we prune unused TSVs. After a burn-in period, the algorithm ends with a smallest number of TSVs of the target 3-D NoC.

The communication cost of an application onto regular, irregular or custom 3-D NoC is represented as in Equation 9.6. We consider any type of a 3-D NoC as an irregular or custom 2-D NoC; we extract the distance matrix of the target 3-D NoC as explained in Figure 9.3. Then, we calculate the communication cost of a current configuration using Equation 9.6.

In Figure 9.17, communication costs of PFMAP and NMAP algorithms on differ-

Input: Number of task nodes for an application (N)

Output: X, Y, Z for 3-D NoC design

Function getDimensions (N)
$X = Y = Z = \lfloor \sqrt[3]{N} \rfloor$
while $X * Y * Z < N$ **do**
    $X++$;
    if $X * Y * Z \geq N$ **then**
        $break$;
    end if
    $Y++$;
    if $X * Y * Z \geq N$ **then**
        $break$;
    end if
    $Z++$;
end while
RETURN X, Y, Z
EndFunction

Figure 9.16. Determine X, Y, Z dimensions for 3-D NoC.

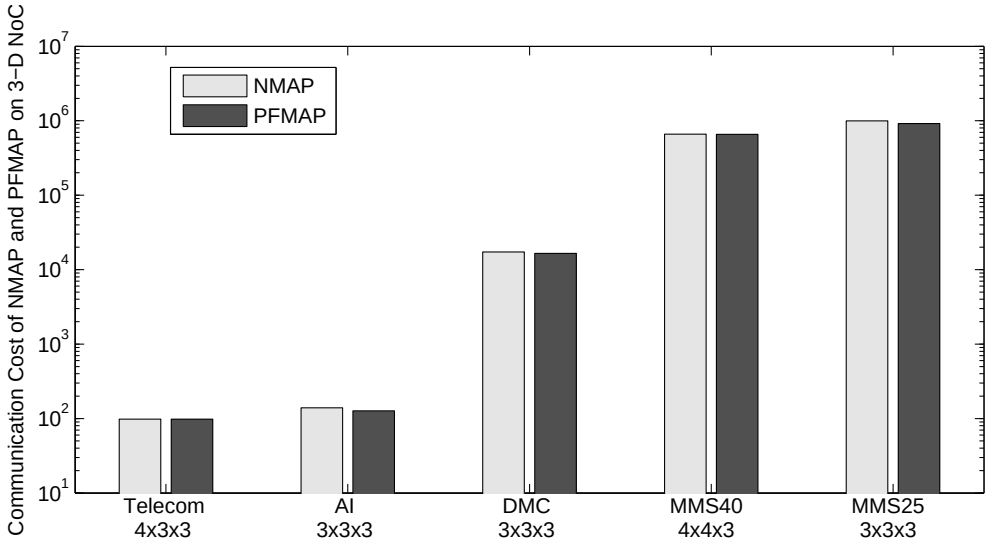


Figure 9.17. Communication cost of PFMAP and NMAP on a 3-D NoC.

ent size of 3-D NoCs for various real life applications are shown. Here, each application is independent from each other. However, it is remarkable that PFMAP tends to give much better results than NMAP when the density of the corresponding task graph is high.

Mapping on 3-D NoCs is also done according to communication energy consumption. For this purpose, energy model given in [94] is used. Here, the average energy consumption of sending one bit of data from *tile* t_i to *tile* t_j is represented as follows:

$$E_{bit}^{t_i,t_j} = nE_{Rbit} + n_H E_{LHbit} + n_V E_{LVbit} \tag{9.13}$$

where, n is the number of routers, n_H number of horizontal links, n_V number of vertical links, all passed by packets. n, n_H and n_V change with the mapping. E_{Rbit} is the energy consumed by a router, E_{LHbit} and E_{LVbit} are the energy consumed on the horizontal and vertical links. All E_{Rbit}, E_{LHbit} and E_{LVbit} values are technology dependent; they can be used as constants as in [94]. We define E_{comm3D} as the sum of all communicating node pairs with the communication energy consumption of E^{t_i,t_j} in a benchmark.

In Table 9.7, we compared NMAP and PFMAP for four 3-D NoCs with different sizes. Even with a small number of IT and PN values, PFMAP outperforms NMAP algorithm.

9.2.4. Large-Scale NoCs

For large-scale NoCs, we have observed that creating initial configurations randomly (line 5 in Figure 9.4) causes to increase in IT and PN values to find a good solution. Instead of using pure random initial configurations for both large-scale 2-D and 3-D NoCs, we apply an initialisation step as in NMAP. The pseudo code for our

Table 9.7. Communication energy comparison of NMAP and PFMAP on 3-D NoCs.

Benchmark	**NMAP**		**PFMAP: IT=100, PN=100**	
	E_{comm3D}	$Run\ T.[ms]$	E_{comm3D}	$Run\ T.[ms]$
TGFF3x3x3	5190	14.35	4767	109
TGFF4x4x4	10820	45.84	10319	382
TGFF5x5x5	24937	713	22950	3027
TGFF6x6x6	53463	2660	44538	16477
Ratio	1	1	0.87	5.82

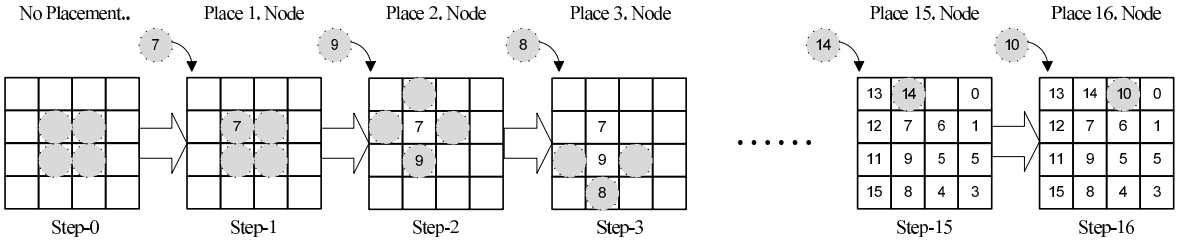


Figure 9.18. PFMAP initialization steps for large-scale NoCs.

initialize method is given in Figure 9.19.

The main difference between NMAP's initialization method and ours is that in each step we select the placement of a node randomly among best candidates with equal costs. In NMAP, the selection operation always finds the same best location. As we find different best candidates with equal cost, we run *Initialization Function* as the number of particles times. Thus, we are able to create cost efficient initial configurations. In Figure 9.18, the initialization steps of a configuration for VOPD application are represented. Yet, we apply initialization only for large-scale NoCs. According to Figure 9.19: in the first step (line 2), we select the task node with maximum communication demand(node 7 in Figure 9.18). Then for the placement, one of the best candidate locations is selected randomly (line 3). The best candidate locations for node 7 are shown as shaded circles in $step-0$ in Figure 9.18. For each initial configuration (i.e. particle), we select one of these best candidate locations randomly. In each step of the initialization phase, there might multiple choices which results in different configurations. For example, in $step-3$, while for one configuration the best location for node 8 is selected as the bottom-neighbor of the node 9, for a different configuration it can be selected as right/left neighbor of the node 9. By selecting best candidates

Input: Set of cores and nodes, TTG, CTG, RCT

Output: Mapping of cores to nodes

1: **Function** initialConf(RCT(R,C))
2: *Select node* N_{max} *with max. comm. demand*
3: *Select one of processor* P_{max} *with max. connection link*
4: *Map* N_{max} *onto* P_{max}
5: **while** *no more node to be mapped* **do**
6: N_{max} = *one of most comm. nodes with placed nodes*
7: P_{max} = *one of procs. which causes min. comm. cost*
8: *Map* N_{max} *onto* P_{max}
9: **end while**
10: **RETURN** $modifiedRCT$
11: **EndFunction**

Figure 9.19. Initialization function.

randomly in this way, we might come up with different configurations. These configurations form our initial configuration set. After obtaining initial configuration set for the given number of particles (first iteration in Figure 9.4), we can apply systematic resampling on this set for the given number of iterations.

We have evaluated the scalability of our PFMAP algorithm on fully synthetic task graphs (generated by TGFF) with various NoC sizes from 3x3 to 9x9 for 2-D regular mesh networks. In Figures 9.20 and 9.21, timing and power results of seven different synthetic task graphs are presented. These synthetic task graphs are mapped onto regular 2-D mesh architectures by using NMAP and our PFMAP algorithm. As already mentioned, we have used NIRGAM NoC simulator for the simulation of each mapping.

As it is evident from Figures 9.20 and 9.21, PFMAP gives lower average packet latency and total power than NMAP independent from the network size. As the network size increases, number of generated packets will increase proportionally. Hence, the gap between network delays of NMAP and PFMAP will rise up significantly. As a

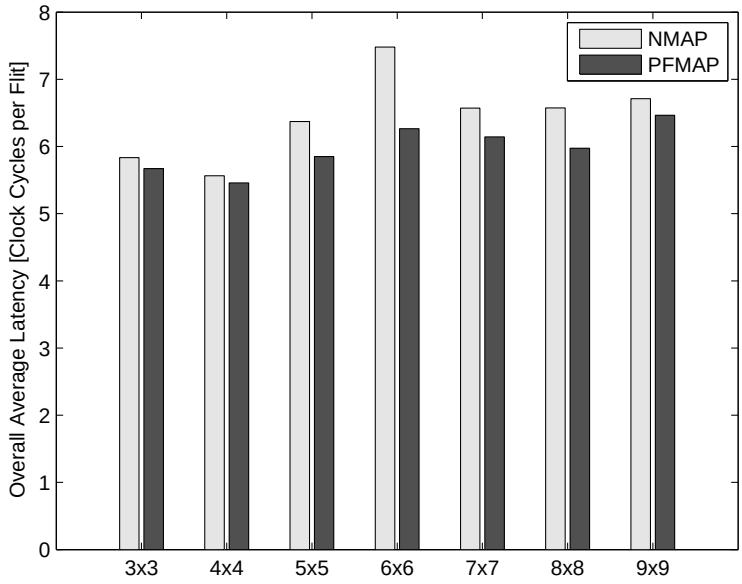


Figure 9.20. Average network latency comparison of NMAP and PFMAP for different size of synthetic task graphs.

result of this, the gap between energy consumptions of NMAP and PFMAP will also increase as the network is getting larger. Finally, we may come up with the result that PFMAP is more scalable than NMAP, since it gives much better results in terms of both total latency and energy consumption as the network getting larger.

9.2.5. Scalability of PFMAP on 3-D NoCs

In our final set of experiments, we have examined the scalability factor of our PFMAP algorithm on 3-D NoCs. In this set of experiments, we have also used fully synthetic task graphs (generated by TGFF) with different number of task nodes (i.e. 27 to 343) and edge weights (i.e. 34 to 521), as we did for 2-D NoCs. In Figure 9.22, communication costs of both PFMAP and NMAP algorithms on 3-D NoC for different size of TGFF applications are available. Here, we also compare PFMAP with itself by setting different ITs and PNs. For example, $PFMAP10x100$, given in Figure 9.22, means corresponding PFMAP solution is found with 10 iterations (IT) and 100 particles (PN). As it is obvious from Figure 9.22, all PFMAP solutions give much better results than NMAP in any network size. It is also clear that communication

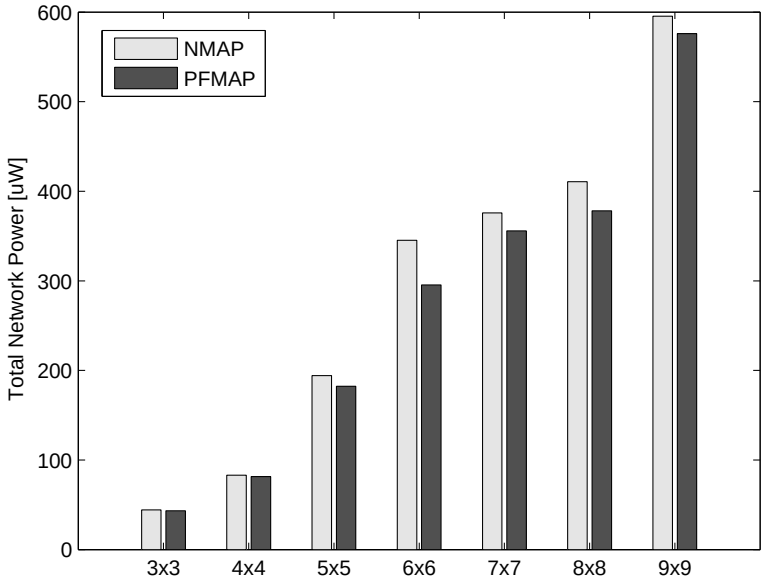


Figure 9.21. Total network power comparison of NMAP and PFMAP for different size of synthetic task graphs.

cost of PFMAP decreases with increasing IT and PN for any network size. The main point here is that PFMAP outweighs NMAP even with a very low IT and PN values in all network sizes.

The corresponding solution finding times of applications in Figure 9.22, are shown in Figure 9.23. It is definite that running times of both NMAP and PFMAP algorithms increase as the problem size getting larger. Similarly, running time of PFMAP algorithm increases for larger IT and PN values in any network size. Although NMAP is a very fast heuristic algorithm, PFMAP10x10 runs faster than NMAP while giving better results than it in all network sizes. We cannot deny that NMAP is a very fast algorithm, but the running time of PFMAP algorithm actually depends on the designer. PFMAP finds a solution in 500 seconds for a huge application with 343 nodes and 541 edges, which gives a better result than NMAP by 20%.

In addition to these, either running time of PFMAP can be reduced or IT and PN values can be increased to find a better mapping solution by using parallel platforms such as GPU.

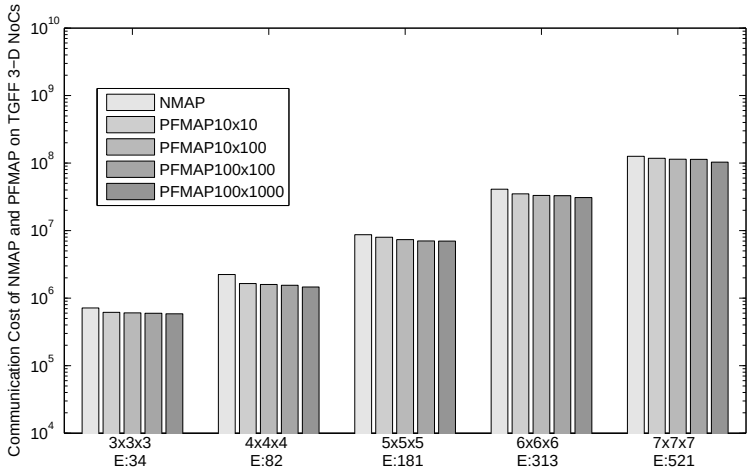


Figure 9.22. Communication cost of PFMAP and NMAP algorithms on 3-D NoC for different size of TGFF applications.

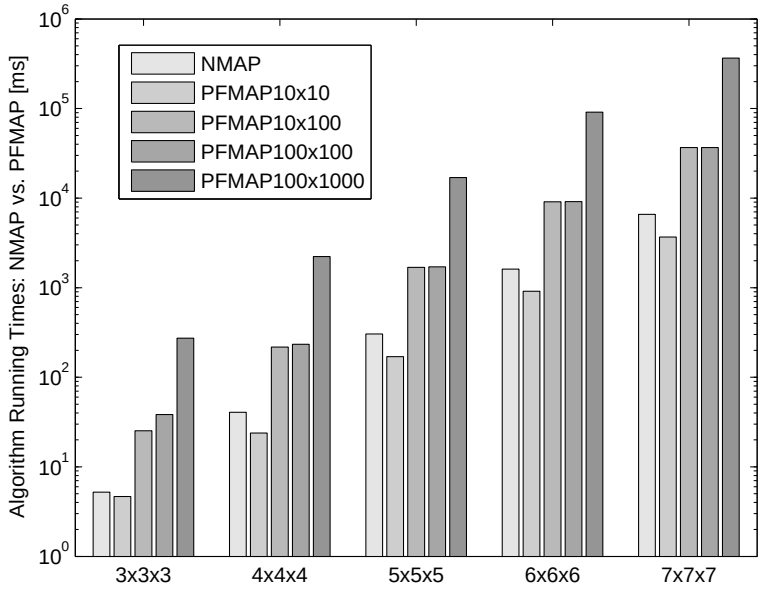


Figure 9.23. Running time of NMAP and PFMAP algorithms with different size of IT and PN for different size of TGFF applications.

10. SIMULTANEOUS MAPPING AND ROUTING FOR NoC WITH PARTICLE FILTERING

The most popular NoC topology is 2-D mesh architecture [52]. In a 2-D mesh NoC architecture, all links have the same length, which eases physical design. Since occupied area of a 2-D mesh topology grows linearly with the number of nodes, it is a scalable architecture. Although 2-D mesh topology is the most common topology, it must be designed in such a way as to avoid traffic accumulating in the center of the mesh. This can be achieved by distributing traffic through the network all the way. To do this, efficient mapping and routing algorithms must be applied to the target topology.

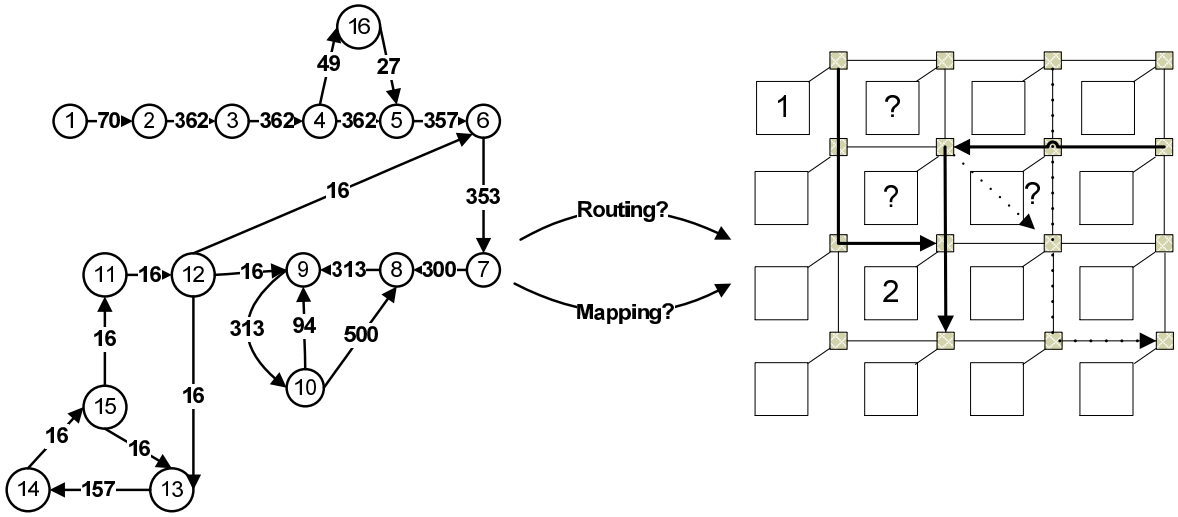


Figure 10.1. Mapping and routing of an application onto a 2-D mesh NoC architecture.

In Figure 10.1, task graph of VOPD application and its mapping with routing on a 2-D mesh topology is given. On the left side of the Figure 10.1, nodes represent the sub-modules of the application and numbers on the edges represent the average communication volume (in 10Kbytes/s) between these modules. On the right side of the Figure 10.1, target 2-D mesh NoC architecture is given. Here, while rectangular white shapes represent processor cores, which are attached to routers, small shaded squares represent the routers and links between these routers show the connectivity of routers for the given architecture. In this scheme, the problem is the mapping of task nodes

onto the physical cores (i.e. mapping problem) and deciding communication paths for these cores (i.e. routing problem). Here, the main objective is to distribute the traffic through the network instead of accumulating it in the center. Mapping process is one to one mapping of each node onto a physical core on the target architecture. As our PFMAP [7] algorithm proposes an efficient mapping, we do not go into detail for mapping process. Routing process is setting communication paths between cores according to the given task graph.

With an efficient mapping and routing, traffic can be distributed through the network by minimizing congestion and contention delays. Congestion and contention delays occur in a network due to

- complex network design,
- inefficient task to core mapping process,
- unsatisfactory routing algorithms.

As a result of these problems, routers become bottleneck of NoC architectures in large networks. In order to avoid routers' negative performance impact to the architecture, congestion and contention latencies must be reduced. The first choice can be using routerless, alternative architectures or reducing the number of routers on the architecture. There are recent works [12–18], which are focused on proposing new NoC architectures by reducing the number of routers on the system. The second choice can be using alternative, efficient mapping and routing algorithms, which decide the physical placement of core pairs neighbour to each other in all way, and thus, route communication requests efficiently by minimizing congestion and contention delays. Third alternative can be a hybrid solution which uses simple switches instead of routers and designing efficient mapping and routing algorithms on that architecture.

10.1. Target Architecture and Main Objective of the Proposed Algorithm

The proposed PFROUT algorithm works on two dimensional reconfigurable NoC architectures. Likewise PFMAP, the input of the algorithm is the task graph of the

single-use case or multi-use cases task graph(s). In these task graphs, nodes represent the tasks and weighted edges represent the communication volume between tasks in 10KBytes/second.

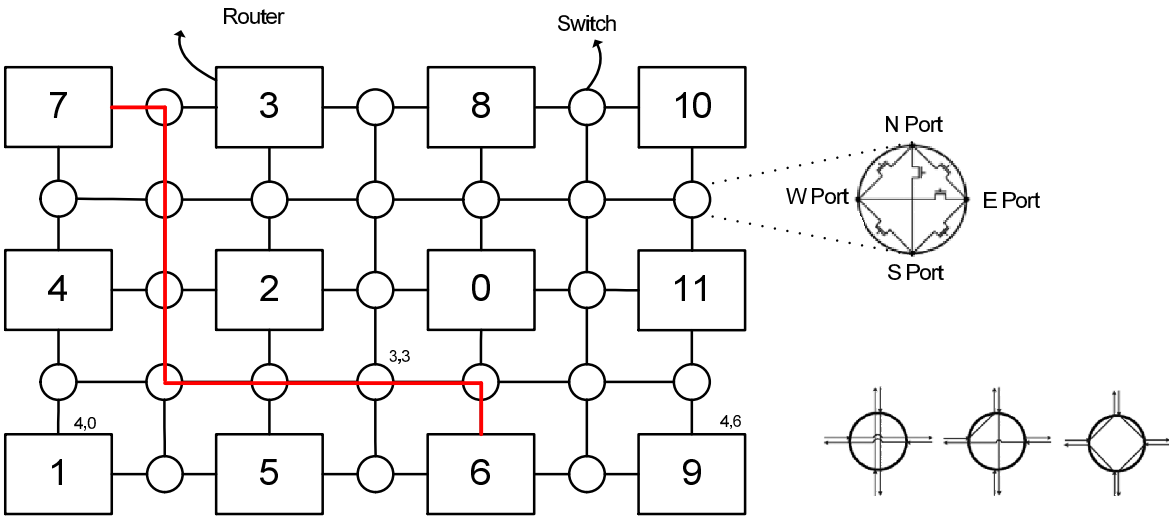


Figure 10.2. 2-D reconfigurable NoC architecture with corridor width one (i.e. CW=1) [1].

The target architecture, which is a mesh based 2-D reconfigurable NoC architecture, is given in Figure 10.2. The given target architecture is inspired from the work proposed in [1]. This architecture has additional simple configuration switches compared to a conventional 2-D mesh NoC architecture. These configurations switches are much simpler than conventional routers. They have only switching capabilities and these switches can be reconfigured at design time, according to the communication requirement of a given application. In Figure 10.2, routers are not connected to each other directly, they are connected to each other through the configuration switches. These switches do not have memory, buffer or any other logic apart from simple switching capabilities.

Properties of simple configuration switches can be given as follows:

- They do not have any memory.
- In each direction (North - East - South - West) they have both inputs and outputs.
- Their switching configuration can be set at design time.
- As they are very simple, they require less hardware and consume less power than

conventional routers.

- As they do not have any memory, they do not support sharing without connection to the conventional routers.

In Figure 10.2, a connection between cores 6 and 7 is set through only simple switches. In this figure, three possible switch configurations in different directions are presented.

The main objectives of the PFROUT algorithm for the target NoC architecture are given as follows:

- Try to set connections through switches.
- If sharing required, use minimum number of routers.
- Find an optimum mapping and routing by placing most communicating nodes close to each other.

PFROUT can find optimum if the traffic flow between nodes is set only through these configuration switches and each node communicates only to its neighbours. By taking all these aspects into consideration, PFROUT tries to find an optimum physical placement of nodes on 2-D Mesh NoC architecture, and find an optimum routing for the communication of the nodes.

The given architecture can be extended by increasing the number of simple configuration switches between routers. In Figure 10.3, the reconfigurable 2-D mesh NoC architecture with the corridor width two (i.e. CW=2) is given. Increasing corridor width would be useful, if there is no solution for a given application for the given corridor width. With increased corridor width, traffic flows have more flexibility to traverse through the configuration switches.

There are two graphs as inputs to the mapping and routing algorithms. Likewise in PFMAP, the first one of these is Task Traffic Graph (TTG), where the task nodes and the communication flows between them are defined. The second input graph is

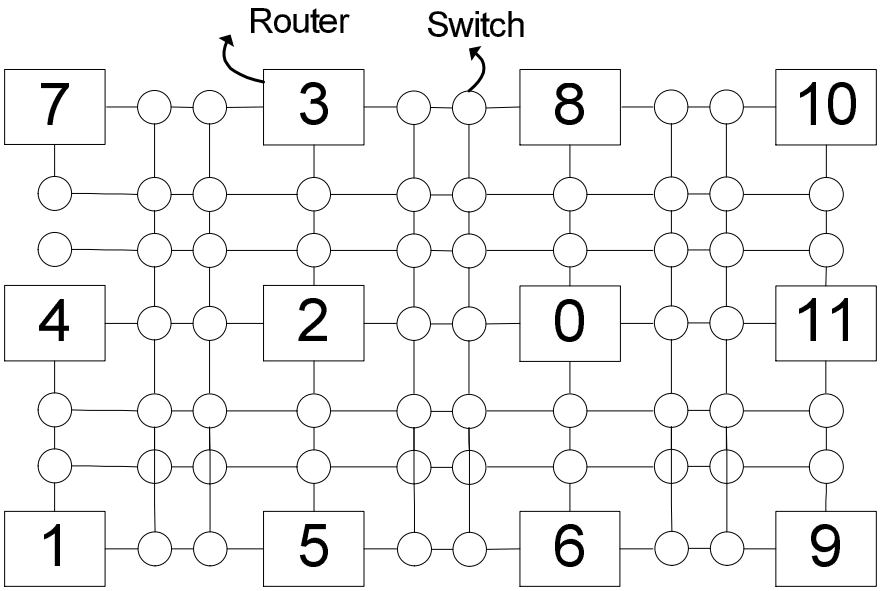
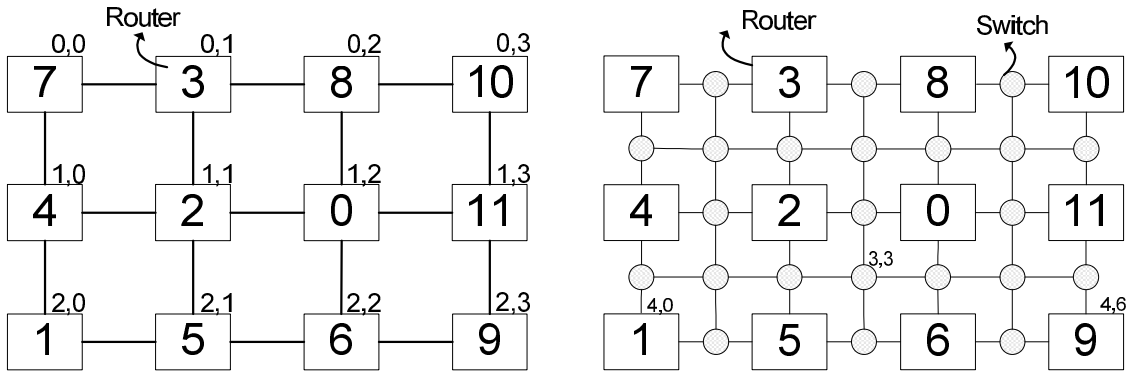


Figure 10.3. 2-D reconfigurable NoC architecture with corridor width two (i.e. CW=2) [1].

called Core Traffic Graph (CTG) in which processor or computational cores and their communication relationships are given. In addition to these, Router-Switch Configuration Topology (RSCT), which defines the location of routers and switches, is also an input to the PFROUT algorithm. Since PFROUT, based on PFMAP algorithm, all definitions and theorems of PFMAP are also valid for PFROUT. There are some additional theorems and definitions for PFROUT:

Definition 10.1. Router-Switch Configuration Topology (RSCT) *is a 2-D reconfigurable mesh NoC topology with each ordered pair* $RS_{rs,cs} \in RSCT(RS, CS)$ *representing either physical location of a switch or processor core attached to a router on the target architecture.* RS *and* CS *indicates the number of rows and columns in the topology respectively (i.e.* $RS \times CS$ *is NoC size).* rs *is the horizontal index number (rowswitch) and* cs *is the vertical index number (colswitch) indicating the physical location of* $RS_{rs,cs}$ *on the 2-D NoC architecture with switches.*

In Figure 10.4, RCT and its corresponding RSCT are illustrated. In Figure 10.4, processor cores are not shown. In both RCT and RSCT, rectangular shapes represent the routers attached to the processor and shaded circles represent the switch nodes. While RCT is composed of only routers, RSCT has both routers and switches. The



(a) An example of RCT(3,4) (b) Corresponding RSCT(5,7)

Figure 10.4. An RCT and its corresponding RSCT.

RSCT is taken from [1]. In RSCT, there are some limitations for routers and switches in terms of their number of inputs and outputs: both switch and router architectures have only n-bit single input and single output in one side (North-East-South-West). Each processor or computational core is attached to a router. In addition to the processor interface, the maximum number of inputs and outputs for both switch and router is four inputs and four outputs. This is illustrated in Figure 10.5. The physical location of router or switch on RSCT defines the number of inputs and outputs for both switch and router. For example, when a router is located on the right upper corner, then it may have only inputs and outputs in West or South direction. In the same way, if a switch resides on the bottom border of RSCT (switch cannot be on the corner) it cannot have any inputs or outputs in South direction.

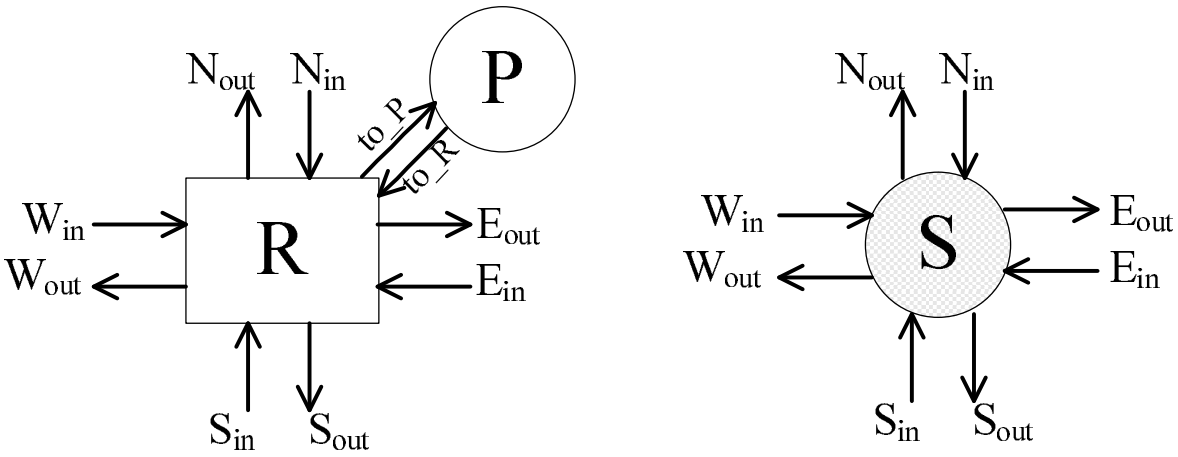


Figure 10.5. Number of available maximum inputs and output for router/switch.

Definition 10.2. Path *is a dedicated link on RCT or RSCT, connecting source and destination nodes travelling through routers and switches.*

The communication cost of a configuration is calculated for mapping and routing by summing cost of each $Path$. In the calculation of $PathCost$, each switch cost on that path is taken as one and each router cost on that path is taken as five. RS_i represents the cost of either a router or a switch on a path. R represents the number of routers and P represents the number of switch on a path. The calculation of $PathCost$ for the source node P_{r_s,c_s} and destination node P_{r_d,c_d} is given in Equation 10.1.

$$PathCost = \sum_{i=1}^{R+S} t_{P_{r_s,c_s},P_{r_d,c_d}} * RS_i \tag{10.1}$$

As mentioned before, $RoutCost$ is calculated by summing $PathCosts$ of each path in a given configuration. The calculation of $RoutCost$ for a given configuration can be found in Equation 10.2.

$$RoutCost = \sum_{i=1}^{E} PathCost_i \tag{10.2}$$

In Figure 10.6, the main part of PFROUT algorithm is given. In the given algorithm each particle represents a configuration. The algorithm generates P random configurations in the first iteration (lines 4-6). Here, we define configuration as the order of cores on tile-based NoC architecture. That is, we randomly place the processor or computational cores on 2-D Mesh NoC architecture initially for each configuration. Here, P is the number of particles and I is the number of iterations, that we define before the beginning of running our algorithm. Since the execution time of our algorithm is proportional to the P and I, we evaluated different values for both of them, e.g. 10, 100, 1000.

Input: RCT(R,C), RSCT(RS,CS), CTG(C,T)

Output: Mapping of cores to nodes and paths between these nodes

```
1: BestFitness = −1
2: for i = 0 to IT do
3:    for j = 0 to PN do
4:       if (i == 0) then
5:          randomConf(particles_j) ;
6:       else
7:          randomNodeSwap(particles_j) ;
8:       end if
9:       Route_j ⇐ findRouting(particles_j, RSCT(RS, CS), CTG(C, T))
10:      particleFitness_j ⇐ Route_j.Fitness ;
11:      CurrFitness ⇐ particleFitness_j ;
12:      if CurrFitness > BestFitness then
13:         BestFitness ⇐ CurrFitness
14:         BestConf ⇐ particles_j
15:         BestRout ⇐ Route_j.Path
16:         BestCost ⇐ Route_j.RoutCost
17:      end if
18:   end for
19:   sysResamp(particleFitnesses, chosens, PN)
20:   chooseResampleds(particles, chosens, PN)
21: end for
```

Figure 10.6. Main part of PFROUT algorithm.

In the first iteration, we find routing for each randomly generated configuration (line 9). The subroutine $findRouting$ is called for each configuration (line 9). For each of these configurations we get fitness values (line 10). The calculation of fitness function for PFROUT is done in $findRouting$ subroutine (see line 17 in Figure 10.7) as given in Equation 10.3.

$$Fitness_{routing} = 1/RoutCost \tag{10.3}$$

According to initial configurations, we calculate the fitness value for each configuration. Among these configurations, the largest fitness value is set as the minimum $BestFitness$ (lines 12-17). After generating initial configurations and finding the *BestFitness*, we re-sample these configurations in each iteration (lines 19-20). Sampling from the distribution and checking those samples with largest fitness values can be utilized for a *low cost RoutCost selection algorithm*. In the remaining IT-1 iterations, we apply pairwise swap among randomly selected nodes (line-7).

In Figure 10.7, the method of finding best routing is given. For each mapping configuration(i.e. particle), this method is called by the main (see line 9 in Figure 10.6). Firstly, for a given mapping $RCT(R, C)$, that is $RSCT(RS, CS)$, traffic amounts t_k are sorted in descending order (line 2). After that iteratively we check the t_k edge weight (i.e. traffic amount) whether it is an edge between two neighbour nodes in $RCT(R, C)$ (lines 5-8). To do this, we call $NeighborHood$ method given in Figure 10.10. If the selected t_k is an is between two neighbour nodes, routing is applied for it (lines 6-7). After routing t_k edges that connect neighbour nodes P_{r_s,c_s} and P_{r_d,c_d}), we apply same routing scheme for non-neighbour t_k edges in descending order (lines 11-16). After that, the fitness function for the given routing is calculated (line 17). At the end of this method, $Fitness$, $RoutCost$ and $Path$ are returned to the main method (see line 9 in Figure 10.6).

In Figure 10.9, the method of finding best routing for each t_k edge is given. This method is called by the method given in Figure 10.7 for each t_k edge. Before starting routing for the current t_k edge, we generate wavefront according to the distances of

Input: RCT(R,C), RSCT(RS,CS), CTG(C,T)

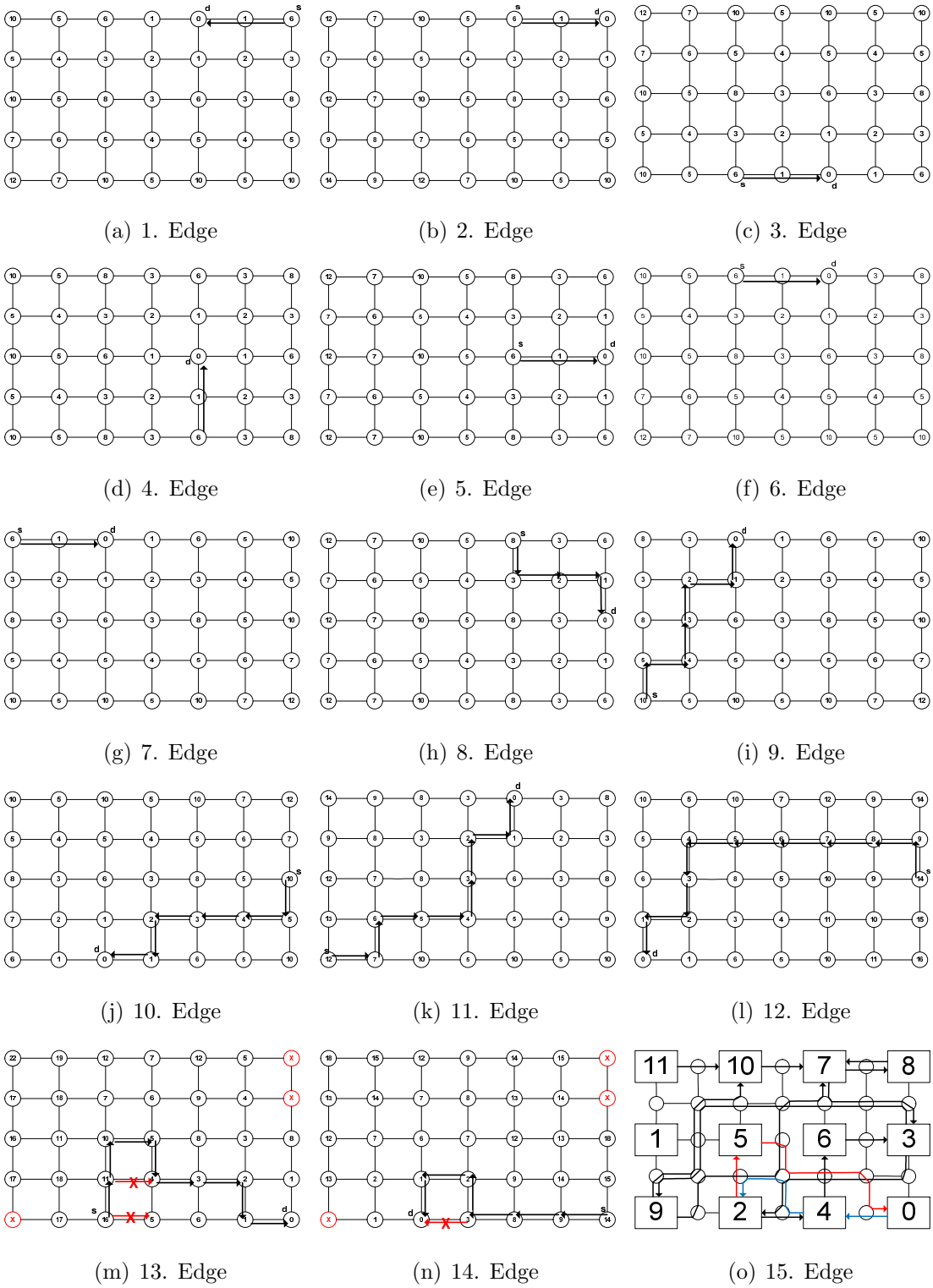
Output: Routing paths between cores in RSCT, Routing Cost

1: **Function** findRouting (RCT(R,C), RSCT(RS,CS), CTG(C,T))
2: *Sort traffic amounts t_k in descending order*
3: **for** $k = 0$ to E **do**
4: $/ * t_k \, is \, edge \, btw \, P_{r_s,c_s} \, and \, P_{r_d,c_d} * /$
5: **if** $isNeighbour(RCT(R,C), P_{r_s,c_s}, P_{r_d,c_d})$ **then**
6: $RoutCost+ = Route(t_k, P_{r_s,c_s}, P_{r_d,c_d}).Cost$
7: $Path[k] = Route(t_k, P_{r_s,c_s}, P_{r_d,c_d}).Path$
8: **end if**
9: **end for**
10: $/ * Remaining \, edges \, routed \, btw. \, non - neigbour \, P_{r_s,c_s} \, and \, P_{r_d,c_d} * /$
11: **for** $k = 0$ to E **do**
12: **if** $!(isNeighbour(RCT(R,C), P_{r_s,c_s}, P_{r_d,c_d}))$ **then**
13: $RoutCost+ = Route(t_k, P_{r_s,c_s}, P_{r_d,c_d}).Cost$
14: $Path[k] = Route(t_k, P_{r_s,c_s}, P_{r_d,c_d}).Path$
15: **end if**
16: **end for**
17: $Fitness = 1/RoutCost$
18: **RETURN** $Fitness$, $RoutCost$ and $Path$
19: **EndFunction**

Figure 10.7. Configuration routing algorithm.

source (i.e. P_{r_s,c_s})and destination (i.e. P_{r_d,c_d}) nodes. In Figure 10.8, there is an example for generating wavefront of a source-destination pair. In the generation process of wavefront, we also consider the capacities of switches and routers. If the current node on $RSCT(RS, CS)$ is a switch, then $t_{i,j}$ value is added to the $RoutCost$ (line 6).

In the similar way, if the current node on $RSCT(RS, CS)$ is a router, then $5 * t_{i,j}$ value is added to the $RoutCost$ (line 8). After this step, the $CurrNode$ is added to the path (line 10). Next, by considering generated wavefront and the capacity of next router/switch, the direction is determined (lines 11). At this step, we always control



(a) 1. Edge (b) 2. Edge (c) 3. Edge

(d) 4. Edge (e) 5. Edge (f) 6. Edge

(g) 7. Edge (h) 8. Edge (i) 9. Edge

(j) 10. Edge (k) 11. Edge (l) 12. Edge

(m) 13. Edge (n) 14. Edge (o) 15. Edge

Figure 10.8. Wavefront generation and routing steps of H-263 decoder for a given configuration.

Input: RCT(R,C), RSCT(RS,CS), CTG(C,T), $t_k, P_{r_s,c_s}, P_{r_d,c_d}$

Output: Route path between cores P_{r_s,c_s}, P_{r_d,c_d}, Route Cost

1: **Function** Route(RCT(R,C), RSCT(RS,CS), CTG(C,T), $t_k, P_{r_s,c_s}, P_{r_d,c_d}$)
2: *Generate wavefront (WF) for source (P_{r_s,c_s}) and destination P_{r_d,c_d} nodes on RSCT*
3: $CurrNode \leftarrow SourceNode$
4: **while** $CurrNode \neq DestNode$ **do**
5: **if** $CurrNode$ is a switch **then**
6: $PathCost+ = t_{i,j}$
7: **else**
8: $PathCost+ = 5 * t_{i,j}$
9: **end if**
10: Add $CurrNode$ to the $Path$
11: *Select $NextNode$ (Switch or Router) with min. cost on generated WF*
12: */* If there is no free direction*/*
13: **if** No available $NextNode$ **then**
14: $NoSolution = SharedPath(t_k, P_{r_s,c_s}, P_{r_d,c_d})$
15: **if** $NoSolution$ **then**
16: *break;*
17: **end if**
18: **end if**
19: $CurrNode \leftarrow NextNode$
20: **end while**
21: **RETURN** $PathCost$ and $Single\,Path$
22: **EndFunction**

Figure 10.9. Routing of single path.

Input: RCT(R,C), P_{r_s,c_s}, P_{r_d,c_d}

Output: P_{r_s,c_s}, P_{r_d,c_d} are (non)neighbour

1: **Function** isNeighbour (RCT(R,C),P_{r_s,c_s}, P_{r_d,c_d})

2: **return** (((${r_s}$==${r_d}$)&& (abs(${c_s}$-${c_d}$) ==1)) ||

3: ((${c_s}$==${c_d}$)&& (abs(${r_s}$-${r_d}$) ==1)))

4: **EndFunction**

Figure 10.10. Nodes neighbourhood function.

the availability of the directions in the order of North-East-South-West.

If there is no direct free direction for the current node, then we call the greedy *SharedPath* method given in Figure 10.11 (line 14). If there is still no available path, even after calling greedy *SharedPath* method, then we break the running of routing algorithm and report as there is no solution found (lines 15-17).

Input: RCT(R,C), RSCT(RS,CS), CTG(C,T), $t_k, P_{r_s,c_s}, P_{r_d,c_d}$

Output: Shared path between cores P_{r_s,c_s}, P_{r_d,c_d}, Shared routing Cost

1: **Function** SharedPath(RCT(R,C), RSCT(RS,CS), CTG(C,T), $t_k, P_{r_s,c_s}, P_{r_d,c_d}$)

2: **if** There is already a path btw. P_{r_s,c_s} and P_{r_d,c_d} **then**

3: Use this path, and calculate single path cost

4: **else if** There is already path from an intermediate node P_{r_i,c_i} to P_{r_d,c_d} **then**

5: Find a new path from P_{r_s,c_s} to P_{r_i,c_i}

6: **else if** There is already path from P_{r_s,c_s} to an intermediate node P_{r_i,c_i} **then**

7: Find a new path from P_{r_i,c_i} to P_{r_d,c_d}

8: **else if** There is an intermediate node P_{r_i,c_i} which can connect P_{r_s,c_s} and P_{r_d,c_d} **then**

9: Find new paths from P_{r_s,c_s} to P_{r_i,c_i} and from P_{r_i,c_i} to P_{r_d,c_d}

10: **else**

11: Report no solution from P_{r_s,c_s} to P_{r_d,c_d}

12: **end if**

13: **RETURN** *SharedPathCost* and *Shared Path*

14: **EndFunction**

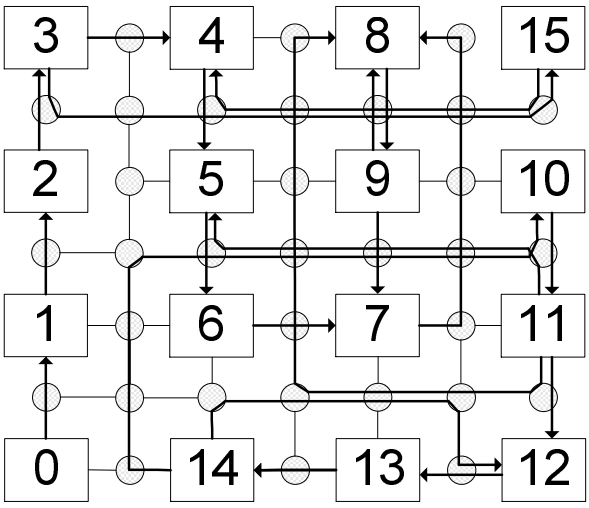
Figure 10.11. Search for shared paths.

In Figure 10.11, greedy shared path approach is given. This method is called by the *Route* method if there is no available direct connection between source P_{r_s,c_s} and destination P_{r_d,c_d} nodes. As searching all possible shared paths is intractable problem, we preferred to use a greedy approach. Hence, backtracking approach, which is time consuming, is not taken into account. At first, we check whether there is already a direct or indirect path between nodes P_{r_s,c_s} and P_{r_d,c_d} (line 2). If there exists such a path, we assign this path to the current t_k edge and calculate this direct or indirect path cost (line 3). If there exists no such a path, we look for an intermediate node P_{r_i,c_i} which has a connection to the destination (line 4). If there is such a path, then we find a new partial path from from P_{r_s,c_s} to P_{r_i,c_i} (line 5). Otherwise, we try to find another intermediate node P_{r_i,c_i} which has a connection from source to itself (line 6). If there is such a path, then we find a new partial path from P_{r_i,c_i} to P_{r_d,c_d} (line 7). Otherwise,in the last trial, we select an intermediate node P_{r_i,c_i} and try to find new paths P_{r_s,c_s} to P_{r_i,c_i} and from P_{r_i,c_i} to P_{r_d,c_d} (lines 8-9). If all aforementioned cases are not possible, we report as there is no any available shared path for the given source P_{r_s,c_s} and destination P_{r_d,c_d} nodes (line 11).

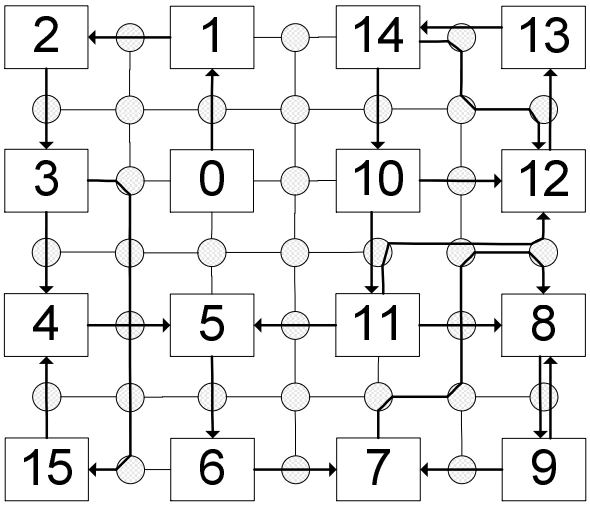
10.2. Case Studies

Likewise PFMAP [7], we implemented our PFROUT algorithm in C++ with OPENMP library [89]. All tests have been carried out on a 32-bit Windows-7 PC with a i5 CPU-750@2.67GHz and 4-GB RAM. We performed our experiments with various applications such as VOPD, MMS-Suite (H263-decoder, H263-Encoder, MP3-Decoder, MP3-encoder), Multi Window Display (MWD), Depth Map Computation (DMC) [95]. In addition to these, we have evaluated the scalability of our PFROUT algorithm on fully synthetic task graphs (generated by TGFF [2]) with various NoC sizes from 3x3 to 10x10 for 2-D mesh networks. Mapping and routing algorithms of PFROUT applied to the regular 2-D Mesh NoC architectures with Corridor Width 1 and 2 (e.g. CW=1, CW=2).

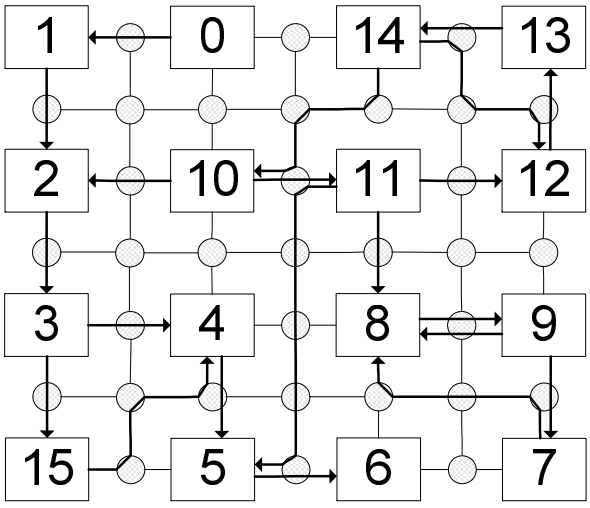
In Figure 10.12, routing of VOPD application with AppAw [1], PFMAP [7] and PFROUT is given. As PFROUT algorithm is a routing dedicated mapping algorithm,



(a) Routing with AppAw [1] *(RoutCost=5753)*



(b) Routing after PFMAP mapping [7] *(RoutCost=5243)*



(c) Routing and mapping with PFROUT *(RoutCost=4539)*

Figure 10.12. Routing of VOPD application with AppAw [1], PFMAP [7] and PFROUT for CW=1.

it gives the best result in terms of *RoutCost*. In Figure 10.12b, routing for PFMAP [7] is given. Here, we apply routing algorithm to the mapping found by PFMAP algorithm. However, PFROUT algorithm (see Figure 10.12c) apply routing for each candidate mapping.

In Table 10.1, *RoutCost* comparison of AppAw [1], PFMAP [7] and PFROUT on MMS-Suite applications is given. MMS-Suite includes applications such as H263-Decoder, H263-Encoder, MP3-Decoder and MP3-Encoder. As these applications use the same set of IP-cores but the traffic pattern among the cores is different for each application, an average graph is used for both mapping and routing process. As explained in AppAw [1], the average graph is composed of all edges values of four input task graphs. Here, both PFROUT and PFMAP outperforms AppAw [1]. Since mapping and routing processes are applied to the average graph of four input task graphs, performance improvements over AppAw [1] for both PFMAP and PFROUT are changeable. As the average task graph of MMS-Suite includes only twelve cores, PFMAP is also capable of finding much better results than AppAw [1]. Both PFMAP and PFROUT reduces *RoutCost* 30% to 35% for H263-Decoder, H263-Encoder and MP3-Encoder. However, for MP3-Decoder application, both PFROUT and routing applied to the PFMAP mapping decreases routing cost up to 79.96%.

Table 10.1. Routing cost comparison of AppAw [1], PFMAP [7] and PFROUT on MMS-Suite application.

Application	**AppAw**	**PFMAP**	**PFROUT**	**PFMAP Imp. over AppAw**	**PFROUT Imp. over AppAw**
H263-Dec	447721	311257	309743	30,48%	30,82%
H263-Enc	628608	409189	409189	34,91%	34,91%
MP3-Dec	212150	42520	42520	79,96%	79,96%
MP3-Enc	272893	181775	181615	33,39%	33,45%

In Table 10.2, *RoutCost* comparison of AppAw [1], PFMAP [7] and PFROUT on MWD, VOPD and DMC applications is given. Here, we also calculated mapping cost of each application for AppAw, PFMAP and PFROUT algorithms. As it is obvious from the table, AppAw algorithm gives the worst results for both mapping and routing whereas PFMAP gives best mappings and PFROUT gives the best routings. Although mappings found by PFMAP are the best ones, PFROUT finds better results

than PFMAP for routing. This shows that a good mapping cannot always be a good candidate for routing. Hence, it makes more sense to run a routing dedicated mapping (i.e. PFROUT) instead of a pure mapping algorithm (i.e. PFMAP). On the right side of Table 10.2, we give PFROUT improvements over both AppAw and PFMAP. PFROUT decreases routing cost up to 48,05% compared to AppAw and up to 34,99% compared to routing, which is applied to the mapping found by PFMAP.

Table 10.2. Routing cost comparison of AppAw [1], PFMAP [7] and PFROUT on MWD, VOPD and DMC applications.

Application	Mapping Cost			Routing Cost			PFROUT Routing Imp.	
	AppAw	PFMAP	PFROUT	AppAw	PFMAP	PFROUT	Over AppAw	Over PFMAP
MWD	1248	1216	1248	1632	1504	1376	15,69%	8,51%
VOPD	4265	4125	4135	5753	5243	4539	21,10%	13,43%
DMC	14203	12393	12926	114858	59666	38787	66,23%	34,99%

10.2.1. Comparison of PFROUT Routing performance with AppAw [1] on Synthetic Graphs from TGFF [2]

In this set of experiments we compare AppAw [1] and PFROUT routing algorithms on synthetic graphs generated by TGFF [2]. In order to achieve a fair comparison, we generated 10 graphs for each of the network sizes from 3x3 to 10x10. Here, network size represents the dimension of the target 2-D mesh architecture in terms of row and column numbers. As, there are 100 graphs for each scenario, we always consider the average values for both routing algorithms. For PFROUT, we tested each scenario for different number of iterations and particle numbers. For example, PFROUT10x100 means, PFROUT algorithm is used with 10 iterations and 100 particles.

In Figure 10.13, solution percentages of PFROUT and AppAw [1] algorithms on TGFF graphs with the dimensions from 3x3 to 10x10 are given. As the routing problem is intractable and both routing algorithms are heuristic approaches, they cannot find a suitable routing for a given task graph and the target architecture. In Figures 10.13(a) and 10.13(b), we see all direct (i.e. ithout sharing path) or indirect (i.e. with shared path). In PFROUT, it is very easy to find a new routing for a given scenario. If

there is no solution, a new routing can be tested by increasing number of particles and iterations of the algorithm. According to these results, we can say that number of solutions found by PFROUT is more than the algorithm given in AppAw [1] study. In general it is clear that number of solutions decreases for both algorithms as the network size and corresponding task graph becomes more complicated (e.g. having more nodes and edges). For the network with corridor width 1, independent from the iteration (IT) and particle numbers, PFROUT finds more solutions than its counterpart in all cases. Moreover, for the network with corridor width 2, all variations of PFROUT always finds solutions independent from the network size. However, AppAw [1] algorithm cannot find all solutions as the network size is getting larger. Similarly, In Figures 10.13(c) and 10.13(d), solution percentages without shared path are given. As it is obvious from these charts, both algorithm tends to use more shared paths as the networks are getting larger. However, PFROUT outperforms AppAw [1] by finding more direct solutions.

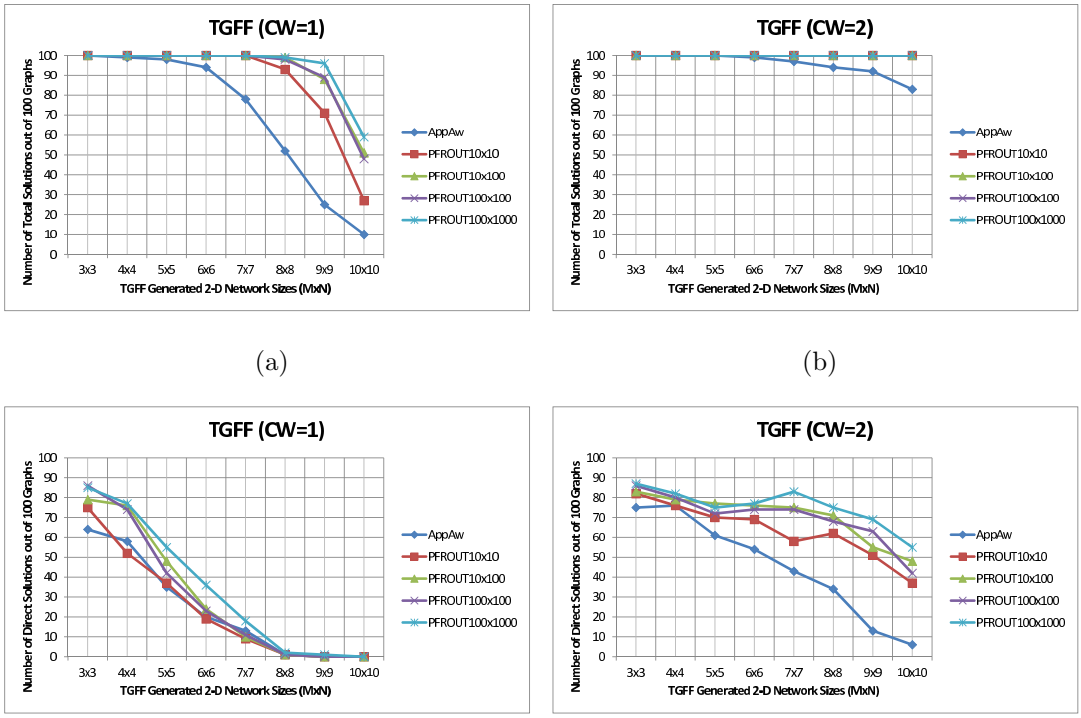


Figure 10.13. Total (shared and non-shared) and direct (non-shared) solution percentages of synthetic task graphs from 3x3 to 10x10 (TGFF).

In Figure 10.14, we compared routing costs of both algorithms for corridor width 1 (i.e. CW=1) on different network sizes from 3x3 to 10x10. Here, PFROUT seems

to give worse results than AppAw [1] for NoC sizes greater than 6x6. As the solution finding percentage of PFROUT is more than AppAw [1] routing algorithm for larger networks, we included only the results where each algorithm finds a solution.As AppAw [1] algorithm cannot find more solutions than PFROUT for NoC sizes greater than 6x6, only the average of solutions that are found are taken into account.

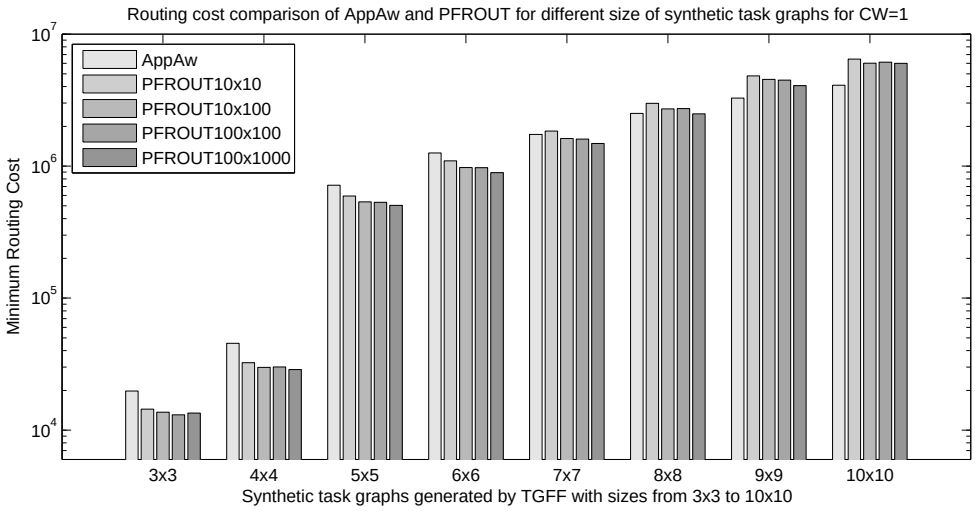


Figure 10.14. Minimum routing cost values of both PFROUT and AppAw [1] algorithms on various networks for CW=1.

Similarly, in Figure 10.15 (for the networks with corridor width 2), PFROUT gives always less routing cost values than AppAw [1] algorithm. Here, it is obvious that routing costs decreases as the number of iterations and particle numbers increase for PFROUT algorithm.

In addition to the abovementioned comparisons, we also compared average number of routers used by two algorithms for different size of networks for both corridor widths one and two (i.e. CW=1 and CW=2). As given in Figures 10.16 and 10.17, number of routers used for both algorithms tend to increase while network size is growing. In all scenarios up to network size 7x7, all variations of PFROUT use less routers than AppAw [1]. As PFROUT finds more solutions than AppAw [1] for large networks, it also uses more routers in that networks.

For corridor width of two, independent from the iteration and particle numbers PFROUT is capable of finding solutions for all network sizes (see Figure 10.13b).

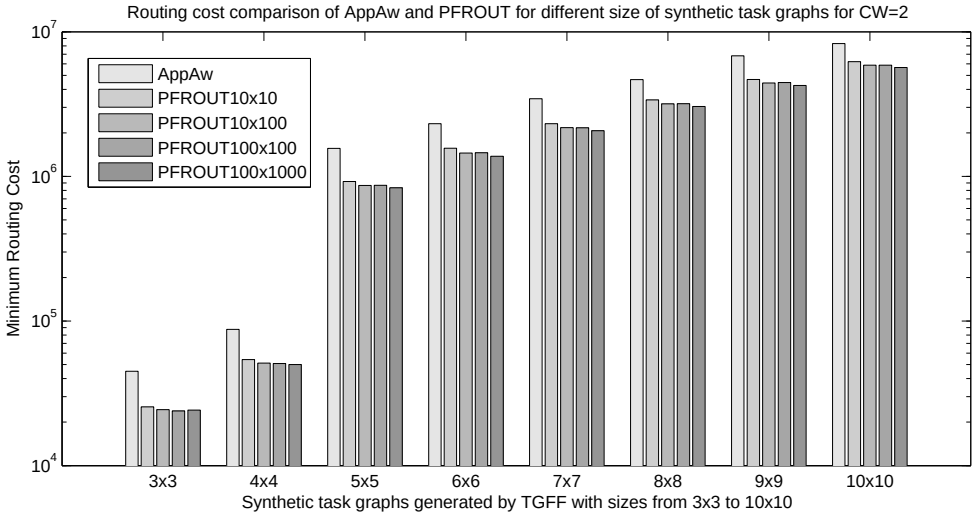


Figure 10.15. Minimum routing cost values of both PFROUT and AppAw [1] algorithms on various networks for CW=2.

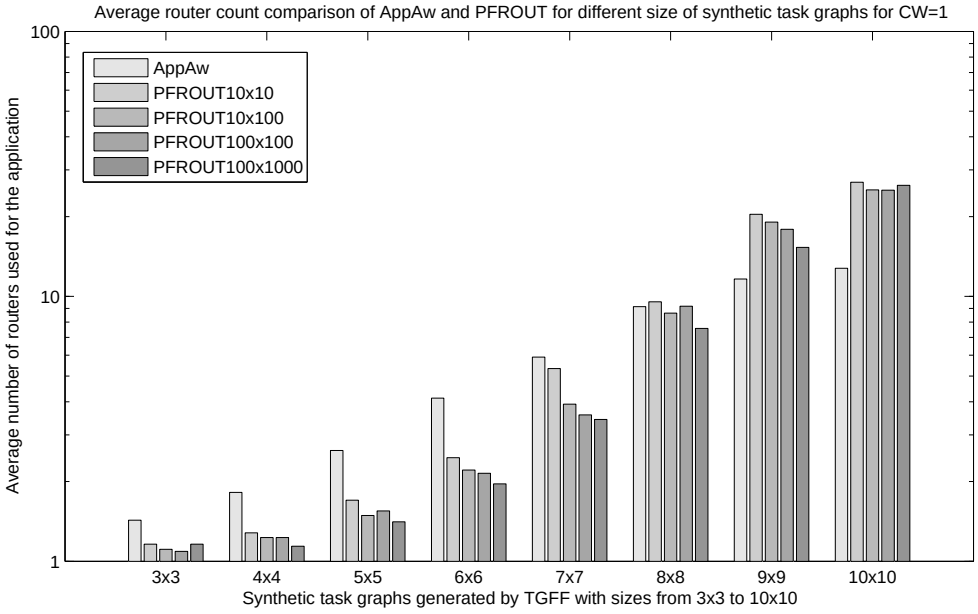


Figure 10.16. Average number of routers used for both PFROUT and AppAw [1] algorithms on various networks for CW=1.

Although, PFROUT finds all solutions for CW=2, it also uses less routers than AppAw [1] for all network sizes. This can be seen obviously from the Figure 10.17.

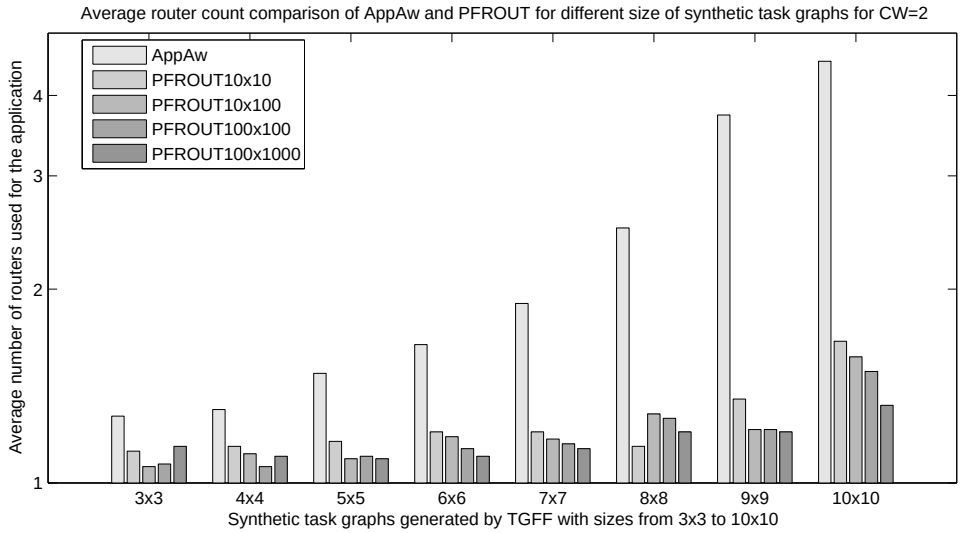


Figure 10.17. Average number of routers used for both PFROUT and AppAw [1] algorithms on various networks for CW=2.

As a last experiment set for TGFF generated graphs, we also tested average running time of both algorithms for different network sizes and CW=1 and CW=2. The running timesof both algorithms for CW=1 is presented in Figure 10.18. Here, the fastest PFROUT algorithm (with 10 iterations and particle numbers) runs as AppAw [1] up to network size 7x7, while the other PFROUT variations takes more time than AppAw [1]. It is already known that AppAw [1] algorithm is based on NMAP mapping, which is very fast but inefficient. Although, the running time of PFROUT is more than AppAw [1], it finds a solution in ten seconds for the worst case scenario (for the network size 10x10, with 100 iterations and 1000 particles).

Similarly, in Figure 10.19, algorithm running times for both AppAw [1] and PFROUT is given. As we already discussed previously, PFROUT is slower than AppAw [1] in general. This is valid also for CW=2 value. However, for also CW=2, PFROUT is capable of finding better results than AppAw [1] with a very little timing overhead. Again, in a worst case PFROUT cand find solutions in about ten seconds, which is not much for a static routing algorithm.

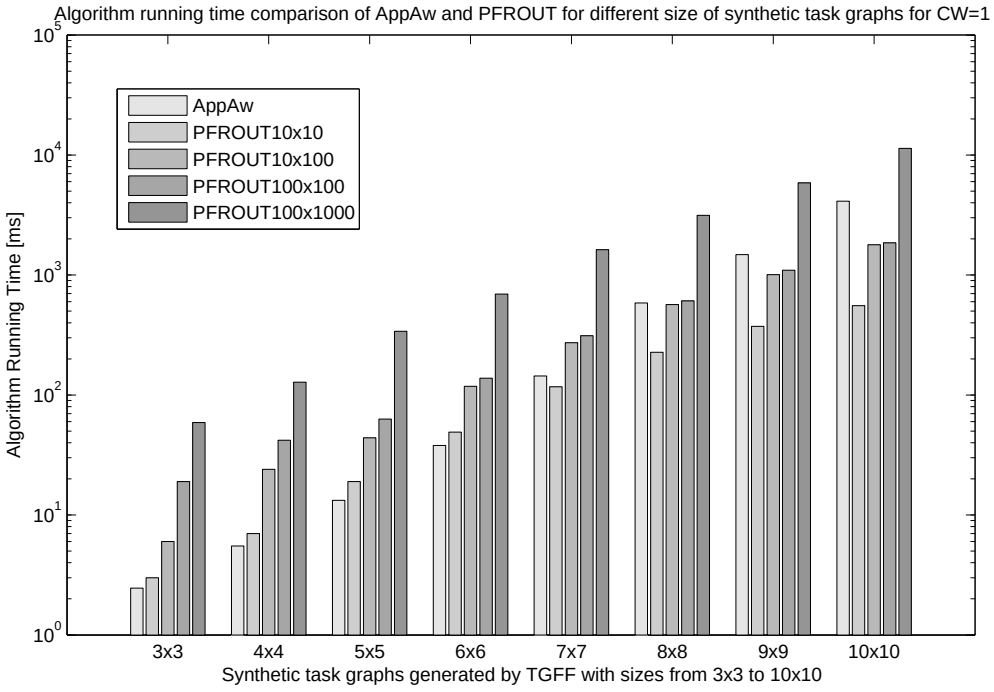


Figure 10.18. Comparison of algorithm running times of both PFROUT and AppAw [1] algorithms on various networks for CW=1.

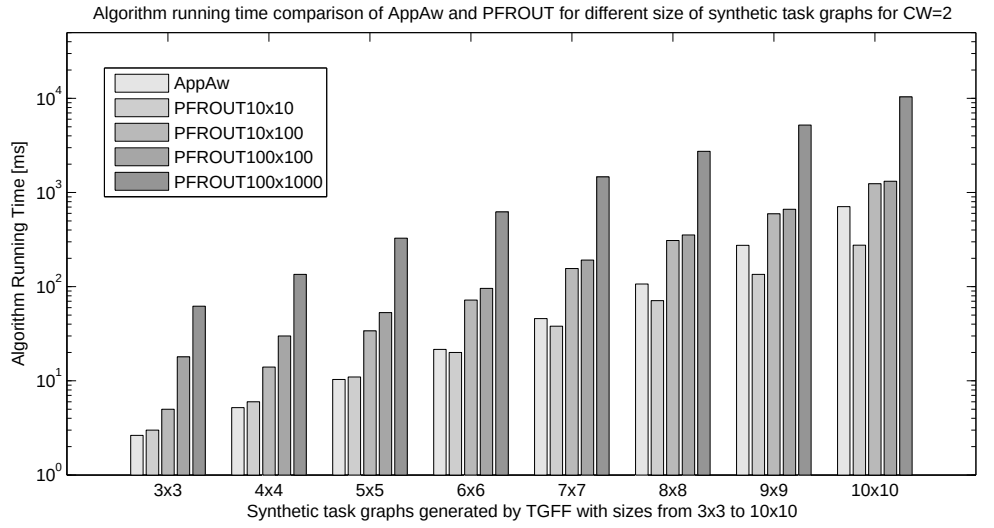


Figure 10.19. Comparison of algorithm running times of both PFROUT and AppAw [1] algorithms on various networks for CW=2.

11. DYNAMIC RECONFIGURABLE POINT-TO-POINT INTERCONNECTS

In this chapter of the thesis, we propose DRP2P interconnects for setting up direct connection between two communicating units before the communication starts. In DRP2P, routers need not exist. Hence, area overhead and power consumption due to routers are completely not present in DRP2P. As a result, the area of the implementation is reduced. This directly reduces power consumption. Direct communication paths are setup by dynamic partial self-reconfiguration. This is done by exploiting the dynamic partial reconfiguration property of the Xilinx FPGAs with c^2PCAP, a dedicated on-chip engine developed for this purpose. This core adds a small overhead in the area and power consumption. However, the experimental results show that power consumption of DRP2P is much lower than that of a traditional 2D NoC.

This is the first work that introduces DRP2P communication architecture. In Section 11.1, we will concentrate on the basic properties of this architecture, compare it with traditional NoC and $k \times k$ crossbar, discuss limitations of the architecture. Then, in Section 11.2, we will explain how DRP2Ps are obtained. This is the section where the self reconfiguration engine, c^2PCAP is also introduced with its applications on not only on low-cost FPGAs like Spartan 3, 6, but also high-end FPGAs like Virtex 4. We will show that with our engine, it will be possible to setup DRP2Ps even on low-cost Xilinx FPGAs. In Section 11.3, we present results of basic tests and the real-life case studies. The final section concludes the work and sets directions for future research.

11.1. Proposed DRP2P Architecture

This architecture is inspired from the fact stated at the first statement of this chapter: P2P is the fastest communication way. In [96], a study has been carried out to specify application-specific point-to-point (ASP2P) interconnects between pairs of

cores during design time. These interconnects remain static during run-time. Our method improves this approach by introducing a different set of P2P between pairs of cores to the SoC in a time-multiplexed manner. We call the set of P2P interconnects in one time-slot as the "communication scenario". Each communication scenario must fit at the related communication channel. During design time, the application is profiled and analyzed so as to determine the communication scenarios at "each time-slot". The duration of the time-slot can be either dictated by the designer or computed by the static analysis. The channels are decided by analyzing and synthesizing the scenarios. In other words, to replace the current communication scenario with another one, dynamic reconfiguration engine c^2PCAP firstly erases the communication channel (tear-down), and then reconfigures the interconnects of the new scenario (set-up).

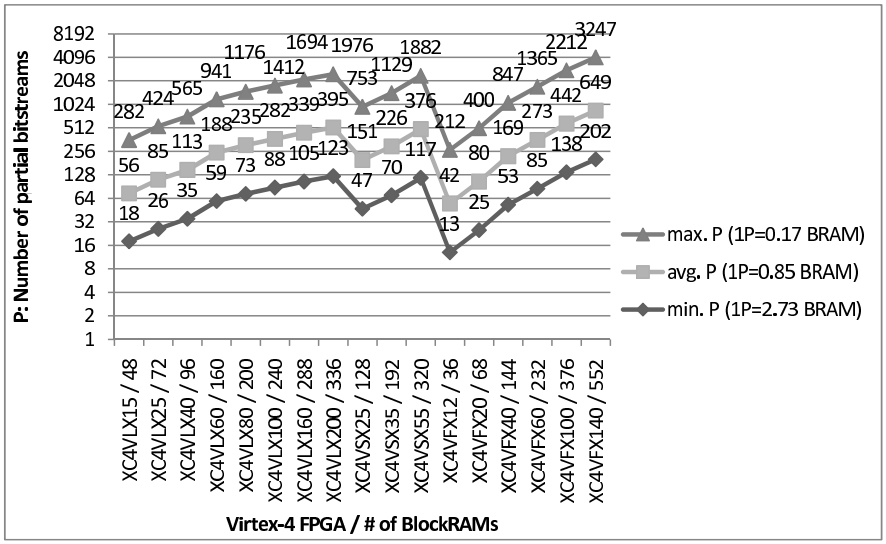


Figure 11.1. Different P (# of partial bitstreams) values that can be stored in the BRAM on Virtex-4 FPGAs.

Since the on-chip memory is limited, infinitely many communication scenarios cannot be realized with DRP2P. The memory reserved for compressed bitstream storage, compression ratio, size of reconfigurable area, device type and size determine the number of communication scenarios. Figure 11.1 gives the information about number of partial bitstreams P that can be stored in the BRAM on Virtex-4 FPGA family. These values are application specific and may vary from application to application.

In addition to the c^2PCAP core, we have also a monitoring system in our design.

The main task of monitoring system is to trigger the c^2PCAP core whenever a valid communication request comes from a module. A communication request can be valid, when it is already predefined and approved by also other modules, otherwise the request is kept waiting until it is approved by the others. That is, when a module is finished with its computation job, it should wait its next communicating partner to finish its computational task. Moreover, a communication grant signal is assigned to each module; these signals are altered only by the monitoring system.

In DRP2P, we reconfigure interconnects. We do not go in detail which ones are control signals to control the flow. These are determined during design time, during this time the analysis of the application is done. Different clock speeds are also supported by DRP2P. If two cores need to communicate at different clock frequencies, either one of the cores can be replaced with a dual port RAM or buffers can be placed in the communication channel. Obviously, cores at different communication speeds would require buffers to prevent data loss. However, we did not experiment this scenario because it is easily do-able with our approach: it should be noted that we are dynamically reconfiguring a region, i.e. the communication channel, not solely P2P interconnects. Therefore different communication architectures can be configured, regardless of topology. The only constraint is the area of the communication channel. We carried out such an implementation in our first case study MTT, where the communication scenarios are not solely P2P but there might also exist architectures like crossbar or broadcast. Hence, if desired, buffers can also be placed equally likely.

DRP2P is designed for nonreactive systems. Start and end times of communication scenarios are not simultaneous. The longest communication time determines the duration of each scenario. However, profiling and analysis of the application has to be carried out before deciding on the communication scenarios.

11.1.1. Theoretical Latency Analysis of DRP2P

Assume that there exist M modules communicating each other and there are N different communication scenarios. Half of these modules ($M/2$) are located on the

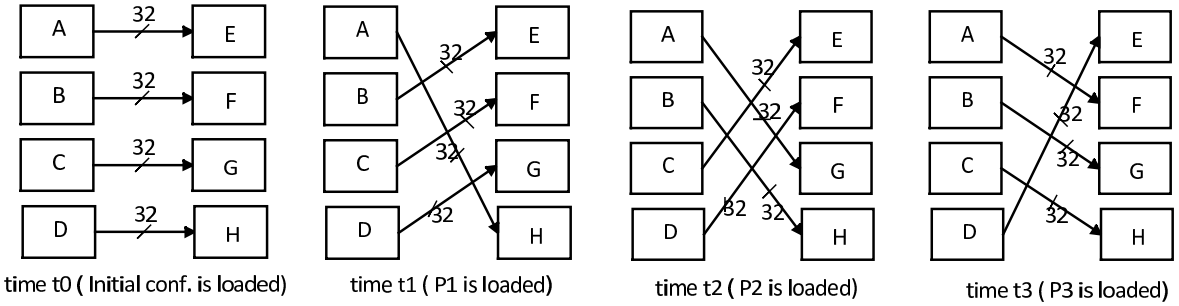


Figure 11.2. Four different communication scenarios.

left side and the other half resides on the right side. The modules in each half are placed one above the other and vertically. Suppose that data flow occurs from left to right and each module on the left half has a directional n-bit connection with its opposite partner on the right half. Hence, the channel width W_C between modules (the total number of single wires between modules on both sides) is $M/2 \times n$. The duration of one communication scenario is given by the longest data transfer time between two communicating modules in that scenario. The data transfer time can be defined as follows: Assume that the data is transferred in k packets and let the size of each packet be s. For the maximum utilization of the wires between these modules, a different portion of data has to be sent at each cycle of the system clock, T_{clk}. Consequently, there should be no idle time until the data are completely transferred to the receiving module. Then, the data transfer time between two communicating modules is given as:

$$T_{comm} = ((k \times s) \div n) \times T_{clk} \tag{11.1}$$

Note that some of these scenarios can be repeated during the execution of the multiprocessor system. Therefore, there can be L ($>= N$) scenarios in an application. Again, for illustrative purposes, we can assume that $L = N$ and each scenario runs only once. Assuming that the data transfer time in each communication scenario is equal, the total communication time is given as:

$$T_{comm\_tot} = N \times (((k \times s) \div n) \times T_{clk}) \tag{11.2}$$

If we assume that the initial communication scenario is downloaded with the complete bitstream, we can say that the same system requires $N-1$ dynamic reconfigurations during the runtime of the multiprocessor system. Let RRR (Reconfiguration Repetition Rate) define the number of dynamic reconfigurations. There can be multiple locations for DRP2P interconnects and the required area has to be reserved during design time. Therefore the dynamic reconfiguration time, T_{reconf} for each reserved area can be easily determined. If we assume that there is only one reserved area for DRP2P, then we can calculate the total reconfiguration time given as:

$$T_{reconf\_tot} = RRR \times T_{reconf} \tag{11.3}$$

Now, referring to Figure 1.3 and Equation 11.1, we can write the best case and worst case conditions for DRP2P interconnects as follows:

$$T_{best} = T_{comm} = ((k \times s) \div n) \times T_{clk} \tag{11.4}$$

$$T_{worst} = T_{comm} + T_{reconf} = ((k \times s) \div n) \times T_{clk} + T_{reconf} \tag{11.5}$$

Similarly, the best and worst case conditions for the total data transfer time can be calculated by using Equations 11.2 and 11.3:

$$T_{best\_tot} = T_{comm\_tot} = N \times (((k \times s) \div n) \times T_{clk}) \tag{11.6}$$

$$\begin{aligned} T_{worst\_tot} &= T_{comm\_tot} + T_{reconf\_tot} \\ &= N \times (((k \times s) \div n) \times T_{clk}) + RRR \times T_{reconf} \end{aligned} \tag{11.7}$$

The data transfer size S_D in each scenario and the total size of transferred data S_{D_tot} in the entire system are given as:

$$S_D = (M \div 2) \times k \times s \tag{11.8}$$

$$S_{D\_tot} = N \times S_D = N \times ((M \div 2) \times k \times s) \tag{11.9}$$

Based on the derived equations on a sample instance, we can numerically demonstrate the latency due to DRP2P interconnects. An implementation of a 32-bit width ($n = 32$), 8 modules ($M = 8$) communication structure ($W_C = 4 * 32$) is illustrated in Figure 11.2. As we have 4 different communication scenarios ($N = 4$) in Figure 11.2, there exists 3 passes between scenarios. Hence, RRR is 3 for this example. Assume that packet size s is 4-bytes(=32-bits). The target FPGA is Virtex-4SX35 and the system clock is set to 100MHz ($T_{clk} = 10ns$). For the reconfiguration, the configuration interface ICAP in 8-bit mode (a byte is sent to ICAP in each configuration clock cycle) is selected and the configuration clock is set to 100MHz ($T_{cclk} = 10ns$); the reconfiguration speed is therefore 100MB/s. Each partial bitstream size is 6524 Bytes. Hence, the reconfiguration time for each pass between different scenarios $T_{reconf} = 6524 * 10ns = 65.24\mu s$. For example, according to the Equation 11.7, for

k=5, the total data transfer time for $S_{D\_tot} = 320Bytes$ in best and worst case (see Figure 11.3) are calculated as follows:

$$T_{best\_tot} = 4 \times (((5 \times 32) \div 32) \times 10ns) = 0,0002ms$$
$$T_{worst\_tot} = 4 \times (((5 \times 32) \div 32) \times 10ns) + 3 \times 0.06524ms = 0,196ms$$

11.1.2. Comparison of DRP2P with 2-D Mesh NoC

For the comparison of DPRP2P with NoC architecture, we have preferred to use the Network on Chip emulator (NoCem) with mesh topology because of its availability. NoCem is available for free download by OpenCores [97]. The NoCem is an open source (protected under GNU General Public License), on Chip Network Emulation Tool and a body of VHDL code configurable by a top-level package file that can create a variety of Network on Chips on parameters of data-width, virtual channel implementations, topology, and in-network buffering lengths. Once parameterized, the resulting NoC is generated automatically with use of VHDL generics and generate statements. It supports three different NoC topologies: mesh, torus and double torus. For a 4*4 mesh topology with 16-bit data width, it occupies 78% of Virtex-II FPGA (xc2vp30) [98].

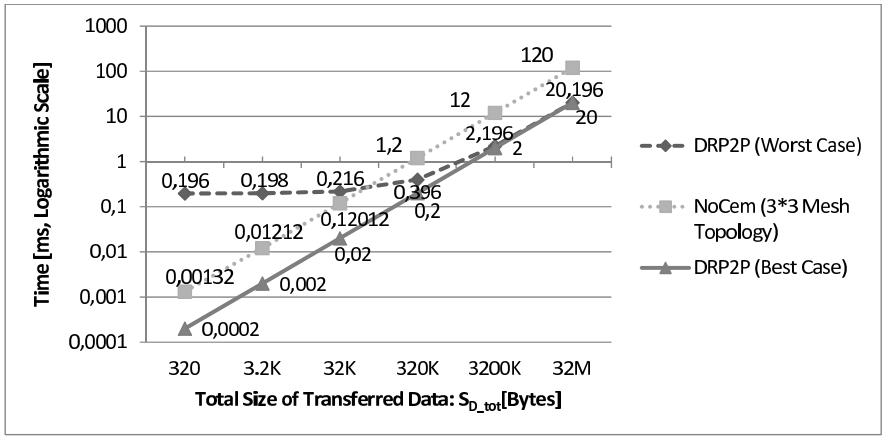


Figure 11.3. Comparison of DRP2P and NoCem with 3*3 mesh topology for different S_D values.

To utilize the DRP2P communication architecture efficiently, the designer should be aware of the fact that the computation time must be longer than reconfiguration latency ($T_{comp} \geq T_{reconf}$). For this example the reconfiguration latency is approximately

about 0,196ms (at 100MHz, 8-bit configuration interface). Here it is obvious that the reconfiguration latency may be decreased to half (98μs) or one fourth (49μs) of the original one by increasing the reconfiguration interface width to 16-bit or 32-bit.

By changing values of S_{D_tot} for this example, we can obtain a comparison of DRP2P with NoCem architecture which is shown in Figure 11.3. We have configured to NoCem as 3*3 mesh topology with simple packet type; with no virtual channels. Here, in the best case, the partial reconfiguration always and totally overlaps with the computation. If the duration of a computation is greater than the partial reconfiguration time of the next communication pattern, then the DRP2P interconnects is the best, regardless of the type and size of the NoC. So, each pattern will be configured during computation, that is, prior to the related communication and hence there will be no reconfiguration overhead in terms of time. However, in the worst case, the computation takes very little time; the time interval of computation never overlaps with the partial reconfiguration.

We did not carry out a specific study on throughput because we transfer data at each clock period in DRP2P. Since there is no buffer or logic between two communicating nodes in DRP2P, the throughput in both cases of is identical to the throughput of a wire. Similarly in experiments with NoC, virtual channel depth is set to 0, i.e. there are no buffers. We carried out all experiments with 100MHz communication and computation speed.

As it is obvious from the chart, in the worst case analysis for the small S_{D_tot} values (up to 320K), the NoC outperforms the DRP2P. However, as the S_{D_tot} value increases (from 320K), the DRP2P gives better results than NoC architecture. Hence, it can be easily claimed that DRP2P architecture is more suitable than NoC approach for large data transfer where the possible number of different communication architecture patterns (N) between nodes are limited in terms of data storage. However, in the best case, DRP2P always outperforms NoC.

Table 11.1 presents the power consumption results of ML402 board and occu-

pied area of the design in Figure 11.2 on Virtex-4SX35 and Spartan-6 XC6SLX45 respectively. We have implemented this design with DRP2P and 3*3 mesh topology with simple packet type of NoCem. Here, the modules M1-M4 are 8*8, while M5-M8 are 16*16 unsigned multipliers for both DRP2P and NoCem designs. The occupied area also includes these LUT-based multiplier blocks. The difference between design types "DRP2P without reconfiguration" and "DRP2P with reconfiguration" is the case, where we have cut the clock input of the ICAP module for the design without reconfiguration. The rest of the designs are identical. The aim is to observe the pure power consumption due to partial reconfiguration.

Table 11.1. Power Consumption of ML402 board and occupied area on Virtex-4SX35 and Spartan-6 XC6SLX45, 8-Modules (W_C:128).

	ML402 Board Power [Watt]	**Virtex-4SX35 occupied area**		**Spartan-6 XC6SLX45 occupied area**	
		Slices	**BRAM**	**Slices**	**BRAM**
Available on FPGA		10086	192	6822	116
Empty Design	3,195	-	-	-	-
DRP2P without Reconfiguration	3,390	2011, (13%)	8, (4.1%)	603, (8%)	8, (6.8%)
DRP2P with Reconfiguration	3,551	2011, (13%)	8, (4.1%)	603, (8%)	8, (6.8%)
3*3 Mesh of NoCem	3,726	10086, (65%)	-	3868, (56%)	-

As it is obvious from the timing, power and area results of DRP2P and NoCem designs, the DRP2P approach gives better results than NoC architecture when the packets being sent are getting larger. In addition to timing performance of DRP2P, it is also less power consuming, more area efficient than the design with 2D-Mesh implementation of NoCem on both Virtex-4 and Spartan-6 FPGAs. When the difference of DRP2P and NoCem with empty design board power is considered, NoCem consumes 531mw, for the same design DRP2P consumes only 356mW; power gain in DRP2P is about 33% compared to NoCem. Therefore, it makes more sense to use DRP2P for such communication infrastructures like in Figure 11.2.

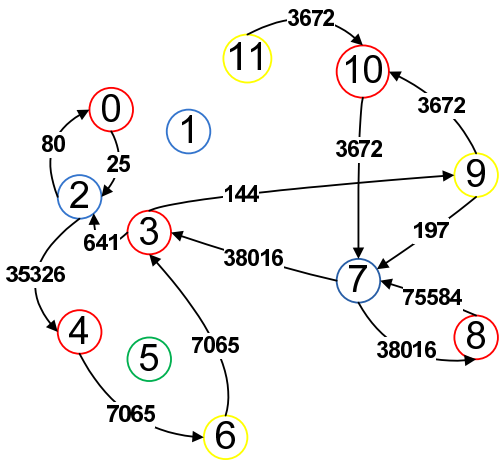
11.1.3. Comparison of DRP2P with a 2-D Reconfigurable Mesh NoC [1]

We have implemented the communication infrastructure of the multi-media system (MMS) by utilizing runtime partial reconfiguration property of Spartan-6 FPGA.

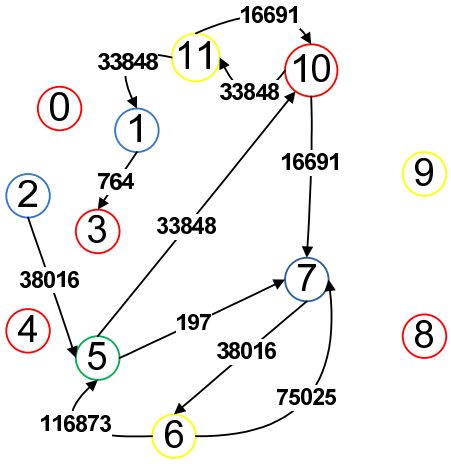
The MMS benchmark includes H.263 decoder, H.263 encoder, MP3 encoder, and MP3 decoder applications [1]. Although the communication flow among the cores is different for each application, each application uses the similar set of IP-cores. In Figure 11.4, the task graph of each application in MMS suite is given. Here, it is worth to mention that each application uses not all cores but only some of them in the MMS suite. Therefore, unused cores (nodes that have no connections to any other nodes in the each task graph) for each application can be switched off to reduce the power consumption. This can be done by pruning the clock input of unused core at each scenario change, i.e. application switching. The pruning process of clock input, which is named *clock gating* method, for unused core can be done by utilizing partial runtime reconfiguration.

As in [1], we have also used the same task mapping procedure for MMS suite. In this scheme, the different tasks of MMS are mapped on 12 cores. In [1], MMS is implemented on the reconfigurable NoC, which uses not only routers but also configuration switches for packet routing.

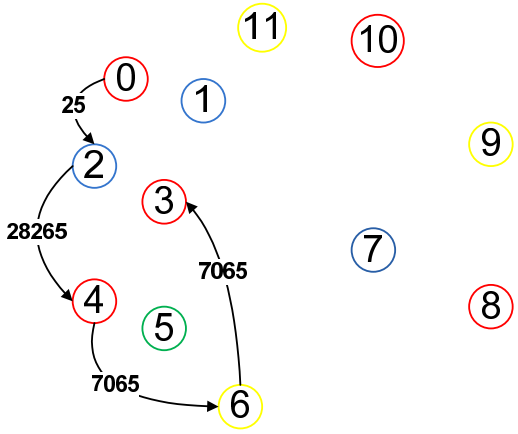
The switching time between most use-cases in a SoC is of the order of few milliseconds [99]. According to the work in [1], the switching operation between applications in MMS can be done either by a configuration manager in reconfiguration process or by storing configuration data in configuration switches and routers. In our case, the switching operation between these applications is done through runtime partial reconfiguration of Spartan-6 FPGA. We have defined the fixed area for cores, which are common all 4 applications in MMS, and the reconfigurable area for the P2P interconnects, which are different for each application. This is shown in detail in Figure 11.5. In our approach, there are dedicated P2P 32-bit links for each communication flow in MMS task graph. The runtime reconfiguration latency and power dissipation for configuration switching can be ignored due to infrequent switching [1].



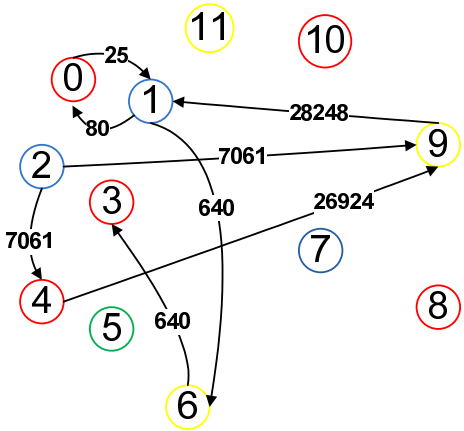
(a) H263-Decoder Task Graph



(b) H263-Encoder Task Graph

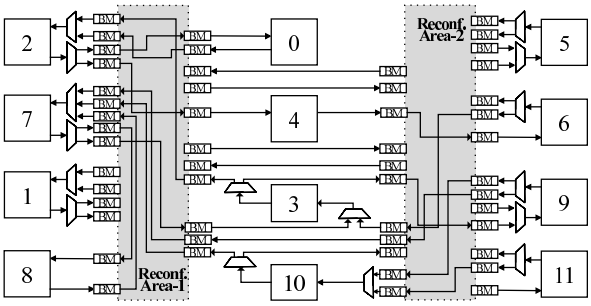


(c) MP3-Decoder Task Graph

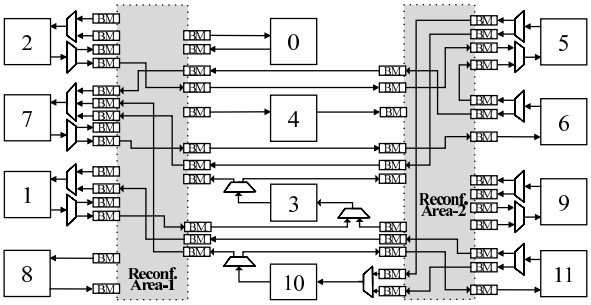


(d) MP3-Encoder Task Graph

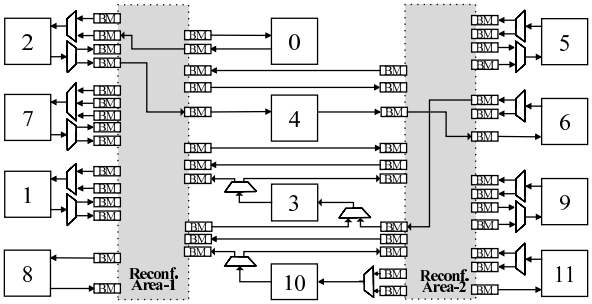
Figure 11.4. Task graphs of applications in MMS suite.



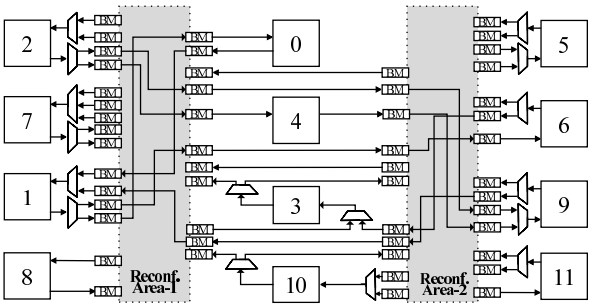
(a) H263-Decoder physical placement on FPGA



(b) H263-Encoder physical placement on FPGA



(c) MP3-Decoder physical placement on FPGA

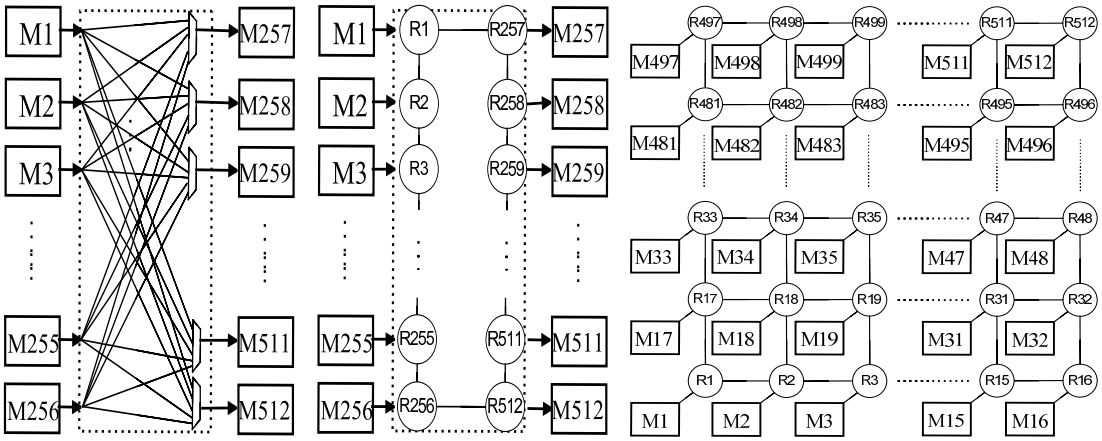


(d) MP3-Encoder physical placement on FPGA

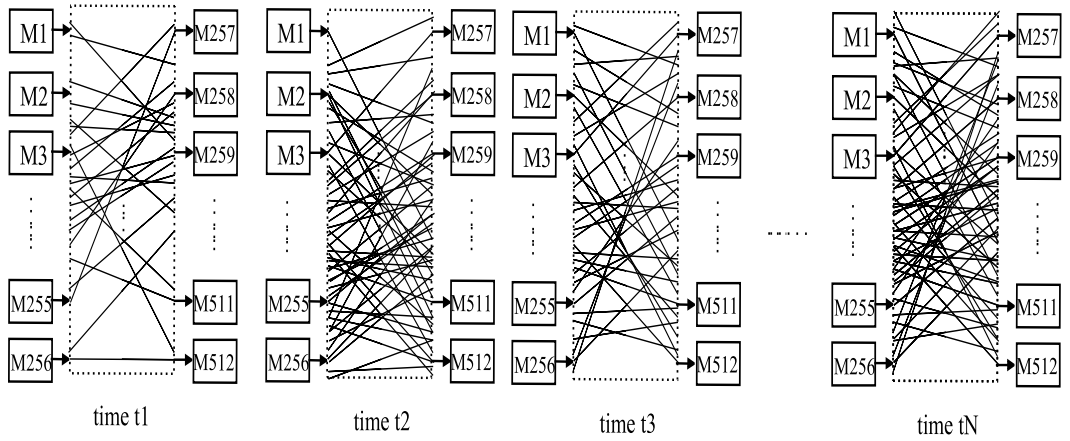
Figure 11.5. MMS physical placement on FPGA.

11.1.4. Comparison of DRP2P with other communication architectures

In Figure 11.6, an n-bit width 256*256 cross point communication architecture and its possible implementations are illustrated. The first and the most common implementation of this architecture is multiplexer based approach, shown in Figure 11.6a. It cannot be denied that this approach is the most speed efficient. Switching between different communication patterns occur in a few clock cycles. However, the occupied area by this approach is infeasible. Even for 1-bit width of 256*256 cross point architecture, the required number of slices is 19476 /20480 of a Spartan-XC3S2000 (95% of the whole chip). Hence, the power consumption of such a huge circuit will be very high. Therefore this approach will not be practicable for small FPGAs (<2M gates).



(a) Mux-based Implementation (b) NoC-Ring Topology (c) NoC-Mesh Topology



(d) N possible communication patterns

Figure 11.6. n-bit width k*k (k=256) point communication architecture and possible implementations.

The second possible way to implement this architecture may be using a ring Network-on-Chip (Figure 11.6b). However, it is obvious that communication from one node to other mostly requires multi-hops (e.g. M1 to M512 requires 256 hops) which actually results in increased latency due to packetization, routing and switching through multiple routers. Apart from having increased latency, this approach seems to be infeasible in terms of area and power.

Alternatively, a mesh NoC, Figure 11.6c, can be a competitor to other approaches. Like the multiplexer based approach, the mesh topology is also more time efficient then ring topology NoC. The worst case time cost of communication from any node to any other node (e.g. from M1 to M512) is traveling about 46 routers (vertically 15 hops, horizontally 31 hops). The number of hops can be drastically reduced by adding some new links to the mesh topology. However, because of very complex routing between routers the occupied area by this method is unpractical to be implemented on a small FPGA. Even a 4*4 16-bit width 2D-mesh grid topology occupies about 78% of Virtex-II FPGA (xc2vp30) [98].

All above mentioned approaches suffer from either speed or area (likewise power). A balanced solution in terms of speed and area may be the DRP2P communication architecture, represented in Figure 11.6d. Such a configuration architecture can fit into 192/5120 CLBs of a Spartan-XC3S2000 (3.75% of the whole chip). To pass 256 signals safely, we have used 32 bus macros, which are located one below the other. In addition to the occupied area of slice based bus macros, 2-CLB-width area is reserved for signal flow from F2R bus macros to R2F bus macros. So, totally 6-CLB-width area is occupied for glitch-free signal flow between modules. As a result, to pass 256 signals 32x6 = 192 CLBs are occupied. The information of each communication scenario must be stored as a partial bitstream for this approach.

11.2. Design Flow for DRP2P

The design flow for DRP2P is given in Figure 11.7. This flow is similar to flow given in Figure 7.1, which is general configuration flow for c^{2}PCAP reconfiguration

engine. Since some of these steps are already mentioned in Section 7.1, here, we only focus on steps, which are related to only DRP2P. Figure 11.7, numbers over the boxes represent the appropriate section for each process.

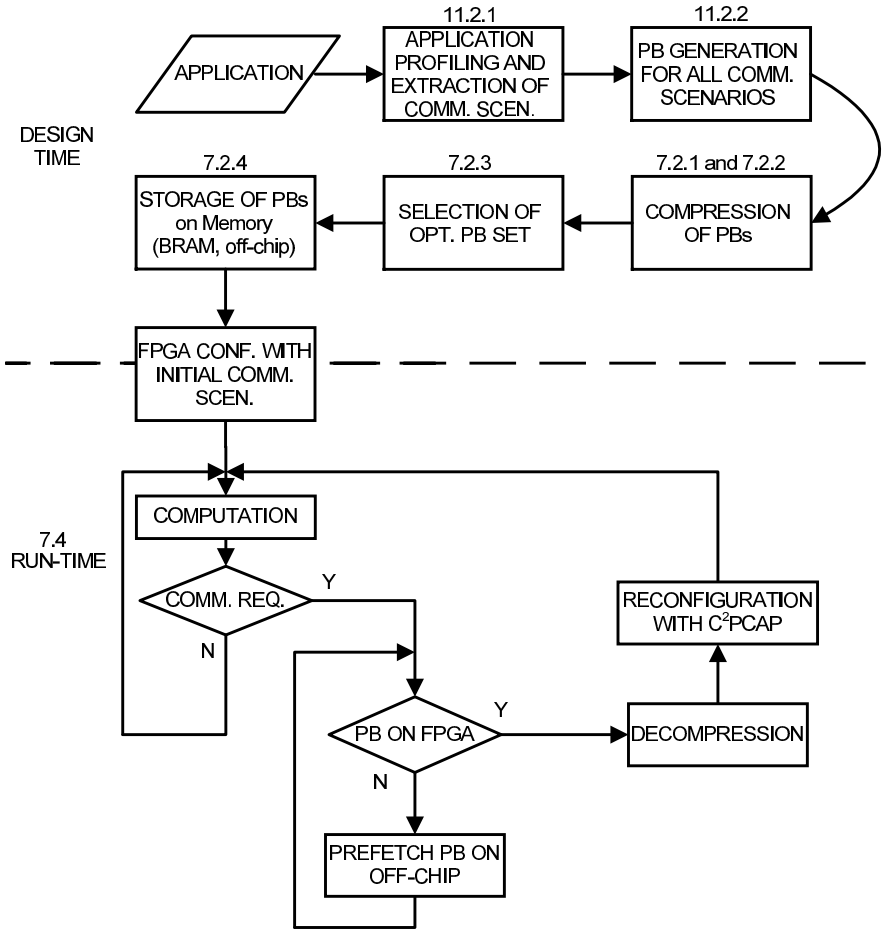


Figure 11.7. Design flow for DRP2P.

11.2.1. Profiling of the Application

Initially, a design on FPGA is being profiled and its communication infrastructure between computing elements is being extracted manually at design time. According to timing and area requirements of the design, the dynamic communication structure of nodes is disclosed with simulation tools. This process gives us the task mappings, communication scenarios and communication channels.

11.2.2. Partial Bitstream Generation for all Communication Scenarios

After determining communication scenarios, a partial bitstream is generated for each of them. Partial bitstream generation steps are given in the following paragraphs.

11.2.2.1. Dynamic Partial Self-Reconfiguration Flow. The c^2PCAP core behaves as if it is a mirror of SelectMAP port, as same in ICAP. The configuration control flow in this work is very similar to *SelectMAP configuration Flow Diagram* in [10] except that PROG, INIT, DONE pins are not taken into account in our study. This control flow has been already illustrated in detail in Figure 5 of our cPCAP core study [9].

Before this, communication infrastructure of the design is located an place, where it communicates through only hard bus macros. The details of generation of hard bus macros are given in the following subsections.

11.2.2.2. Slice Based Bus Macros. For a lossless data communication between reconfigurable and fixed area in a reconfigurable design, the designer must use a type of bus macro, which guarantees a safe data flow in both directions at the time of reconfiguration process. Such a lossless communication can be implemented by tri-state buffers offered by Xilinx Inc. However, some FPGA series from Xilinx such as Virtex-4 and pure Spartan-3 families have no tri-state buffers. Hence, for these device families a new type slice-based bus-macro, which acts as a tri-state buffer, should be developed. There has been some papers published related to this subject such as [100] and [101] using LUT based multiplexer and and-gate approaches respectively. The similar slice based bus macros are used in this work as in Figure 11.8. Contrary to and-gate implementation, transparent latches keep the last data before reconfiguration starts, thus there is no data loss during reconfiguration. To guarantee a glitch-free signal flow from one side to other side two kinds of bus macros (from fixed to reconfigurable area: F2R and from reconfigurable to fixed area: R2F) are used. Since, each bus macro (narrow mode) has a width of 2-CLB, totaly 4-CLB-width area is occupied for F2R and R2F slice based bus macros to pass 8-bit signal from one side to other side safely.

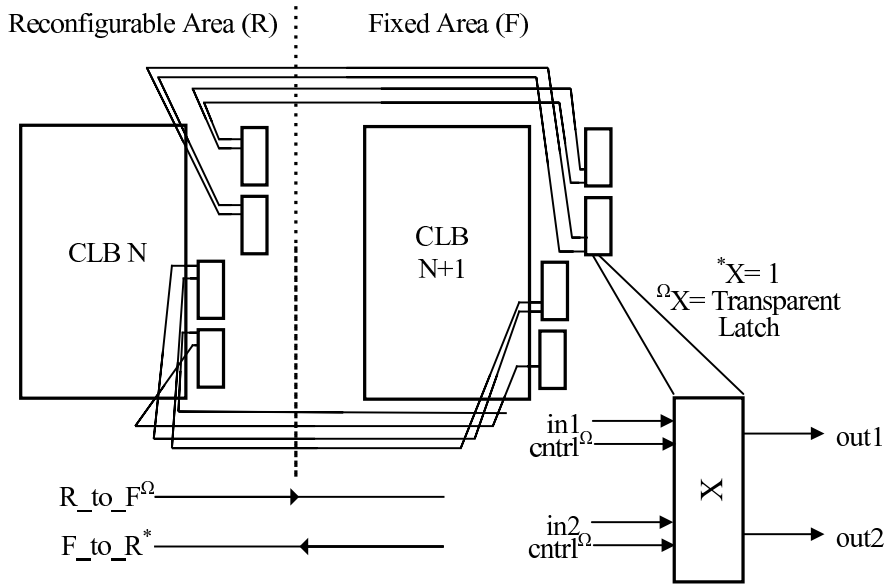


Figure 11.8. General structure of slice based bus macros.

Actually, there are no available slice based hard bus macros offered by Xilinx for Spartan-6 FPGAs. However, there are some bus macros which are directly used for some target FPGA families (e.g. Virtex-5, Virtex-6) from Xilinx. Instead of designing a new bus macro we manipulated and adapted Virtex-5 bus macros to the Spartan-6 bus macros in our recently published study [21]. The 4-bitwidth synchronous single slice hard bus macro component was illustrated in Figures 3 and 4 in [21].

11.3. Test and Results

The soft c^2PCAP core is developed in VHDL. It has been synthesized on Spartan-3S1000 Starter Kit Board, Atlys Spartan-6 FPGA Development Board and also on ML402 (Virtex-4) Evaluation Platform for experimental purposes.

The terms *"Compression Ratio (CR)"* and *"Space Savings (SS)"* used in tables can be formulated in Equations 11.10 and 11.11 respectively (Compressed Size:CS, Original Size:OS).

$$CR = (OS \div CS) : 1 \tag{11.10}$$

$$SS = ((1 - (CS \div OS)) \times 100)\% \tag{11.11}$$

Here it is noted that the *"CS"* and *"SS"* are actually based on the structure and size of the partial bitstream.

Different communication patterns have been designated between a number of computing cores on Spartan-3S1000 and Virtex-4SX35. There are four different communication scenarios in all examples and a partial bitstream is generated for each one; i.e. RRR=4. Note that, the communication scenarios change in round robin manner; i.e. $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \rightarrow p_1 \rightarrow p_2...$

The implementation cost of the communication structure is summarized in Table 11.2 and 11.3. In these tables, various designs with different number of modules(8 to 48) and W_C values (16 to 384) are shown. The reconfigurable area is fixed to 2 CLB columns for every design. In Table 11.2, the size of each original partial bitstream and of all other possible joint bitstreams for three different designs are available. The reconfigurable area for each design is the same and has the same number of logic components. Hence, as the number of modules and also wiring between modules increases, the compression ratio decreases. The reason for this phenomenon is that the number of consecutive zeroes in the partial bitstreams decreases. The communication pattern between modules for each design is completely random and different from each other. The results for the designs with regular connection patterns would be better than our results.

As it is obvious from the tables, by increasing the number of modules, bigger compressed bitstreams are obtained. In Table 11.2, while p_4^c is 744 Bytes for the design with 8-modules (W_C:16) and it is 9742 Bytes for the design with 48-modules (W_C:192). The choice of smallest partial bitstream sets is summarized in Table 11.3. After generating all possible partial and joint bitstreams for three different designs,

Table 11.2. Partial bitstream size and their compression ratios for communication reconfiguration on Spartan-3 1000 FPGA.

Partial Bitstream	**# of Modules**	**Bit Width**	**Original Size**	**Compressed Size**	**Compression Ratio**	**Space Savings**	**# of BRAMs**	**# of BRAMs**
			Bytes	Bytes			Original	Compressed
p_1	8	16	24060	750	32.08:1	96.88%	11.74	0.366
	16	32		1050	22.91:1	95.64%		0.512
	48	384		7523	3.32:1	69.85%		3.673
p_2	8	16	24060	719	33.46:1	**97.01%**	11.74	0.351
	16	32		1070	22.49:1	95.55%		0.522
	48	384		7575	3.18:1	68.52%		3.699
p_3	8	16	24060	743	32.38:1	96.91%	11.74	0.362
	16	32		1117	21.54:1	95.36%		0.545
	48	384		8417	2.86:1	65.02%		4.110
p_4	8	16	24060	744	32.34:1	96.91%	11.74	0.362
	16	32		1098	21.91:1	95.44%		0.536
	48	384		9742	2.47:1	59.51%		4.757
p_{12}	8	16	24060	199	120.09:1	**99.17%**	11.74	0.097
	16	32		428	56.21:1	98.22%		0.208
	48	384		4045	5.95:1	83.19%		1.975
p_{13}	8	16	24060	259	92.9:1	98.92%	11.74	0.126
	16	32		487	49.4:1	97.98%		0.237
	48	384		3937	6.11:1	83.64%		1.922
p_{14}	8	16	24060	206	116.8:1	99.14%	11.74	0.100
	16	32		468	51.41:1	98.05%		0.228
	48	384		6824	3.53:1	71.64%		3.332
p_{23}	8	16	24060	236	101.95:1	**99.02%**	11.74	0.115
	16	32		488	49.3:1	97.97%		0.238
	48	384		5424	4.44:1	77.46%		2.648
p_{24}	8	16	24060	222	108.38:1	**99.08%**	11.74	0.108
	16	32		420	57.29:1	98.25%		0.205
	48	384		6805	3.54:1	71.72%		3.323
p_{34}	8	16	24060	238	101.09:1	99.01%	11.74	0.116
	16	32		375	64.16:1	98.44%		0.183
	48	384		6739	3.57:1	71.99%		3.290

Table 11.3. Partial bitstream storage cost of communication reconfiguration for PCAP [8], cPCAP [9] and c^2PCAP cores.

Method	# of Modules	Bit Width	Reference Bitstream	Bitstreams	Total Size (Bytes)	# of BlockRAMs	Average # of BlockRAMs per 1 bitsream	Space Savings
PCAP [8]	all	all	-	p_1, p_2, p_3, p_4	96240	46.99	11.75	0%
cPCAP [9]	8	16	-	p_1^c, p_2^c, p_3^c, p_4^c	2956	1.44	0.36	96.92%
	16	32			4335	2.12	0.53	95.49%
	48	384			33257	16.24	4.06	65.44%
c^{2}PCAP	8	16	p_2	p_2^c, p_{12}^c, p_{23}^c, p_{24}^c	1376	0.67	0.17	**98.57%**
	16	32	p_4	p_4^c, p_{14}^c, p_{24}^c, p_{34}^c	2361	1.15	0.29	97.54%
	48	384	p_1	p_1^c, p_{12}^c, p_{13}^c, p_{14}^c	22329	10.90	2.73	76.80%

the optimal compressed bitstream sets are selected according to Algorithm 7.2. For example, for the design with 8-modules (W_C:16), the bitstream set p_2^c, p_{12}^c, p_{23}^c, p_{24}^c is found as an optimum solution and the average SS is 98.57% for the complete bitstream set.

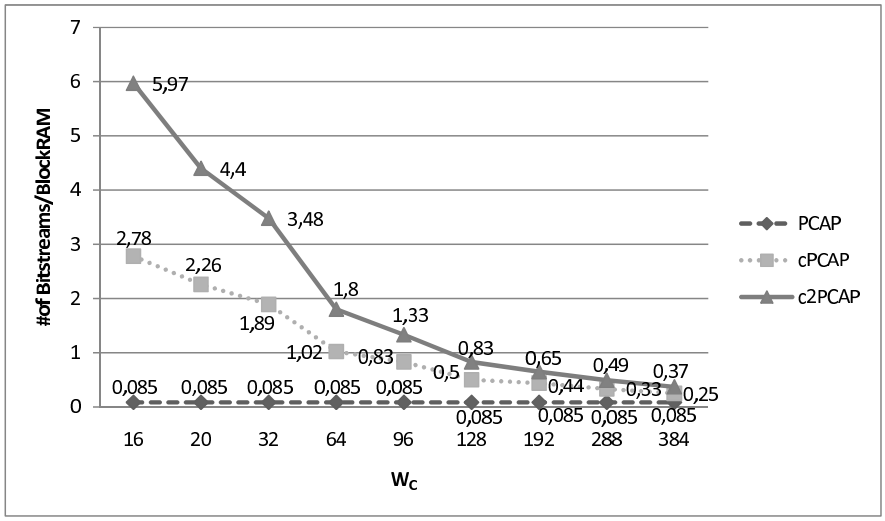


Figure 11.9. # of bitstreams per BRAM vs. W_C for communication reconfiguration of PCAP [8], cPCAP [9] and c^2PCAP cores.

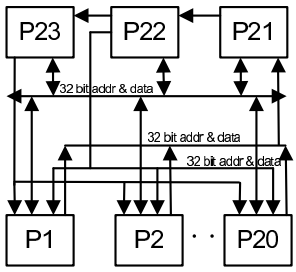
The average number of bitstreams per on-chip BRAM differs from PCAP to cPCAP and c^2PCAP cores. This is illustrated in Figure 11.9. In this chart, it is clear that while the value of W_C is getting larger, the average number of bitstreams per BRAM decreases for cPCAP and c^2PCAP cores. In addition to this, regardless of the design, it is evident that the c^2PCAP core is the most cost effective in terms of storage.

11.3.1. Case Studies

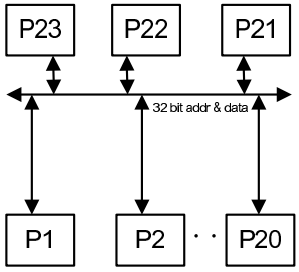
11.3.1.1. Target Tracking Application. A Multiprocessor System-on-Chip (MPSoC) architecture, which is optimized for Multiple Target Tracking (MTT) in automotive applications, is represented in [102]. There are 23 Nios-II processors in this architecture and the communication architecture between them changes with time in a round-robin manner. As illustrated in Figure 11.10, there are mainly five different scenarios that runs sequentially and repeatedly with time. In this architecture, the Pulse Repetition Time (PRT), which is the time interval between two successive radar scans, is 25 ms. Therefore, the total reconfiguration time for five scenarios must be smaller than PRT. In addition to this, the longest reconfiguration time (e.g. 2,88 ms for P2 to P3 at 100MHz with ICAP 8-bit mode) must be smaller than the shortest computation time (8ms for Track Maintenance Block). Since $T_{comp} \geq T_{reconf}$, it is safe to say that the computational time overlaps with the reconfiguration time. Hence, the reconfigurable communication architecture is utilized in an optimum manner. It should be noted that, the communication architecture of this case study is not implemented as DRP2P interconnects. Each communication scenario (Figure 11.10a - 11.10f) is thought as a communication circuitry and replaced with others through DRP2P.

We implemented the communication architecture of this study on a Virtex-4LX100. The reconfiguration latencies for each scenarios are summarized in Table 11.4(a). Although each reconfiguration time in ICAP 8-bit mode at 50MHz is shorter than the shortest computation time (8ms for Track Maintenance Block), due to the fact that the total reconfiguration time (27.17ms) is greater than PRT, it seems to be unapplicable. However, the total reconfiguration latency (13.57ms) using ICAP 8-bit mode at 100MHz is shorter than PRT and each reconfiguration time is shorter than the shortest computation time. Therefore it is feasible to use c^2PCAP with ICAP 8-bit at 100MHz for the best DRP2P interconnects.

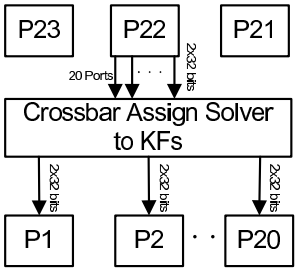
Each uncompressed partial bitstream is approximately 260KBytes for this study. By cPCAP, each partial bitstream is compressed to the 79KBytes approximately with $\sim$70% space savings (storage requires $\sim$40 BRAM blocks). By c^2PCAP, each bitstream



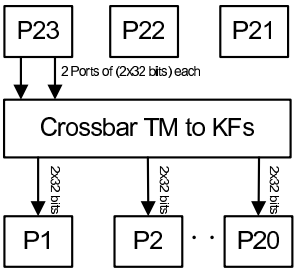
(a) Initial Implementation: 3 shared buses



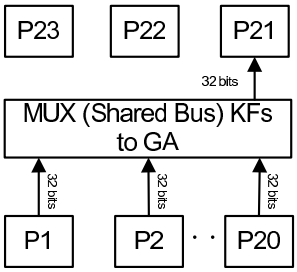
(b) [**P1**]: Shared bus for data and code access



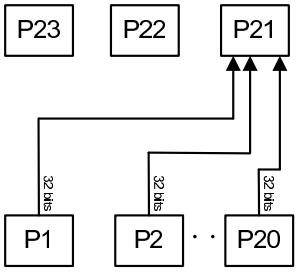
(c) [**P2**]: Crossbar: Assignment Solver to KF



(d) [**P3**]: Crossbar for Track Maintenance to KF communication



(e) [**P4**]: Multiplexer (Bus) for KF to Gating Module Communication



(f) [**P5**]: Multiple low freq busses for KF to Gating Module comm

Figure 11.10. Different scenarios of MPSoC architecture for MTT in a PRT.

is compressed to the 58KBytes approximately with ∼78% space savings (storage requires ∼28 BRAM blocks). As a result, the storage cost of design will be 311KBytes and requires ∼152/240 BRAM blocks on a Virtex-4LX100 ($e.g. p_1, p_3, p_{23}, p_{14}, p_{35}$). By using on-chip memory as a cache, which means storing only bitstream(s) for the next communication scenario and removing past bitstream(s) from BRAM, only ∼60/240 BRAM blocks (25% of total) will be occupied. There are two different approaches to use on-chip memory as a cache, first approach keeps a single bitstream on BRAM while the second one keeps multiple bitstreams. Assume that there is only one partial bitstream and three joint bitstreams ($e.g. p_2, p_{12}, p_{23}, p_{24}$) in the optimal set for four different communication scenarios. In the first approach, only one partial bitstream ($e.g. p_2$) is stored on BRAM, the joint bitstreams are loaded when the corresponding bitstream is needed. With this approach, the storage cost for bitstreams can be reduced by factor $1 \div N$. If we apply this approach to this example (e.g. $p_1, p_3, p_{23}, p_{14}, p_{35}$), we should store at most two partial bitstreams on BRAM, the BRAM usage can be decreased from ∼152/240 to ∼60/240. While it is storage efficient, the main drawback of the first approach is that it may require multiple bitstream load at a time. For

example, assume that only p_1 is available on the cache and p_5 must be downloaded. To achieve this, at first p_1 must be removed then p_3 and p_{35} must be loaded to cache. As a second approach, for the given example (e.g. $p_1, p_3, p_{23}, p_{14}, p_{35}$), p_1 and p_3 are stored on BRAM initially. They can be directly read from cache, when they will be used. To obtain p_2, only p_{23} is taken to cache. For p_4, p_{14} is also loaded to cache, in the same way to get p_5, after removing unused joint bitstreams p_{35} is loaded. So, instead of storing five partial bitstreams on BRAM, it is enough to store at most three of them for this example. So, we may reduce the BRAM usage from ∼152/240 to ∼91/240 for such a design by using cache method.

Table 11.4. Reconfiguration latencies[ms] for case studies.

(a) MTT

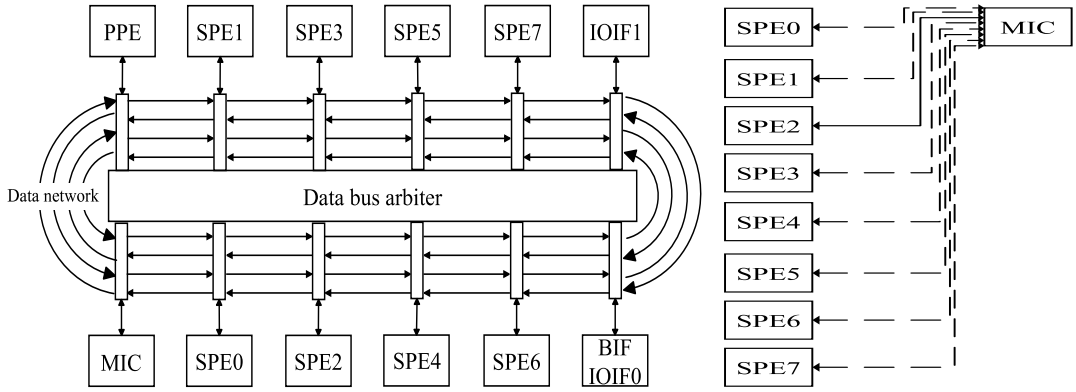
	ICAP (8-Bit)		ICAP (32-Bit)	
Freq.	50	100	75	100
PB				
p_1	5.36	2.68	0.894	0.670
p_2	5.36	2.68	0.894	0.670
p_3	5.77	2.88	0.962	0.721
p_4	5.47	2.73	0.911	0.683
p_5	5.21	2.60	0.868	0.651
Total	27.17	13.57	4.529	3.395

(b) N-Body

	ICAP (8-Bit)		ICAP (32-Bit)	
Freq.	50	100	75	100
PB				
p_1	0.217	0.109	0.036	0.027
p_2	0.217	0.109	0.036	0.027
p_3	0.226	0.113	0.038	0.028
p_4	0.240	0.120	0.040	0.030
p_5	0.226	0.113	0.038	0.028
p_6	0.226	0.113	0.038	0.028
p_7	0.226	0.113	0.038	0.028
p_8	0.324	0.162	0.054	0.040
Total	1.902	0.951	0.317	0.238

11.3.1.2. N-Body Problem. One of the computation intensive problems is the N-body problem [103], which can be applied to extensive applications from various domains in engineering and science. The N-body problem deals with N particles. Each particle interacts with the remaining ones in each time step. To reduce the solution time of this problem ($O(N^2)$), the Barnes-Hut algorithm is applied to the N-body problem and Cell Broadband Engine Architecture is used [104].

In [105], the design of the Cell processors (The Cell Broadband Engine processor)



(a) Element interconnect bus (EIB). (BIF: Broadband interface, IOIF: I/O interface) [105]

(b) 128-bit-width DRP2P communication between SPEs and MIC(Only SPE2 is connected to MIC at this configuration)

Figure 11.11. Element interconnect bus (EIB) and DRP2P communication between SPEs and MIC.

on-chip network and its communication and synchronization protocols are proposed. Figure 11.11a shows the Element Interconnect Bus (EIB), the main component of the Cell processor's communication architecture, which provides the communication between 8 Synergistic Processor Elements (SPEs), Power Processor Element (PPE), I/O Interface and Memory Interface Controller (MIC).

In this case study, we implemented DRP2P to connect the nodes (SPEs) with 128-bit width reconfigurable bus channel to the MIC. These connections change in round robin manner. Hence, there is only one 128-bit-width connection with MIC at a time. So, each SPE communicates (sends or receives, in both directions) with MIC sequentially. Figure 11.11b shows this communication architecture. Here, only SPE2 is connected to MIC, all other SPEs are unconnected and process their computations at this time. While one SPE is communicating with the MIC, the others continue their computations without being interrupted. This architecture is implemented on the Virtex-4LX100 FPGA and the reconfiguration latency for each connection is summarized in Table 11.4(b). Here p_1 is the partial bitstream, which has the communication information of SPE0 and MIC. In the same manner, the rest partial bitstreams have the communication information of SPE1-SPE7 and MIC respectively.

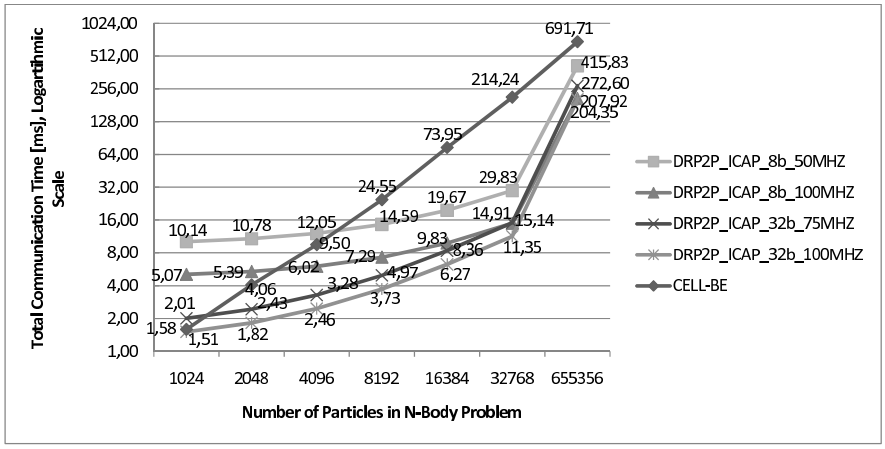


Figure 11.12. Total time consumed for communication between SPEs in N-Body problem with different number of particles.

Figure 11.12 illustrates the communication latencies of EIB on Cell processor and DRP2P interconnects on Virtex-4LX100 FPGA. With a clock speed of 3.2 GHz, the Cell processor is a heterogeneous multi-core chip, which is capable of massive floating-point processing. Our target platform, Virtex-4LX100 FPGA from XILINX, is a reconfigurable, fully parallel platform with a clock speed of 100 MHz. As obvious from the illustration, the DRP2P through ICAP in 32-bit mode at 100MHz (DRP2P_ICAP_32b_100MHZ) has the best performance regardless of the problem's size. The configuration clock speed and the width of configuration interface is proportional to the power consumed during reconfiguration process. Hence, DRP2P with 8-bit at 50MHz (DRP2P_ICAP_8b_50MHZ) can be regarded as the least power consuming method among DRP2P options. On the other hand, it performs worse than EIB on Cell processor when the number of particles fewer than 8192. DRP2P_ICAP_32b_75MHZ and DRP2P_ICAP_8b_100MHZ, are better than Cell processor in terms of communication latencies when the number of particles is greater than 2048 and 4096 respectively.

12. CONCLUSION

In this thesis, we, firstly, discussed dynamic partial self-reconfiguration of Xilinx FPGAs through ICAP and SelectMAP interfaces and storing uncompressed, compressed or joint partial bitstreams on off-chip or BlockRAM within the target FPGA. Instead of using an external intelligent agent, we preferred to use internal agents such as PCAP, cPCAP and c^2PCAP in order to reduce the hardware cost and power consumption simultaneously. As a result of these works, we achieved a very fast partial reconfiguration compared to other serial JTAG interfaces. and using a new approach such as storing modified partial bitstreams on off-chip/ on-chip memory and reading them from there under the control of our efficient reconfiguration engines. The most important advantage of these reconfiguration engines is to achieve very fast partial reconfiguration compared to other serial JTAG interfaces and using a new approach such as storing partial bitstreams on on-chip memory. However, it is obvious that the number of BlockRAM units which are used for storing partial bitstreams cannot be neglected. We overcome this issue by compressing these partial bitstreams in efficient manners. Here, we did not only consider the target partial bitstream itself but also the similarities between other candidate partial bitstreams as well.

This kind of implementation, using no other additional external devices apart from the target FPGA, is a perfect solution for almost all systems based on cost and power consumption. What's also very impressive about this implementation is that the developed reconfigurations can be applied to any type of Xilinx FPGAs. As each of three reconfiguration engines occupies a very small area on the target device, they can be located anywhere in the FPGA, whereas the location of the ICAP module on Virtex-II devices and two ICAP modules on Virtex-4 are fixed [10].

In the following parts of the thesis, we mostly focused on the mapping and routing of task cores onto the nodes of regular, irregular and custom tile-based 2-D and 3-D NoC architectures. Since both mapping and routing problems of NoC are intractable, we preferred to use greedy approaches. To solve both mapping and routing problems on

tile based NoCs, we utilized the systematic resampling algorithm for particle filters. To the best of our knowledge, we are first to apply this algorithm to solve mapping problem on NoC architectures. According to the various experimental results, we showed that both PFMAP and PFROUT algorithms give better results than their rivals. Timing results also showed that both PFMAP and PFROUT are able to find an optimum or near optimum solution in a few milliseconds for medium size commercial applications. We gave the mathematical representations and definitions for both of the algorithms, as well. With scalability analyses, we demonstrated that both algorithms are scalable enough to solve mapping and routing problems larger networks.

Since there is no data dependency between the particles, we applied a multi-thread approach to the parallel running particles. By exploiting C++ OPENMP library, we inserted thread level parallelism to our mapping algorithm. We have already pointed out that the quality of mapping accuracy increases with the available computational resources. Moreover, particles can run in parallel and are very suitable for parallel computation platforms such as GPU, VPU to gain more speed. Thus, heuristic PFMAP and PFROUT algorithms can run much faster on fully parallel platforms and therefore gives better results in shorter times.

We also proposed dynamic reconfigurable point-to-point interconnects in parallel multi-core architectures. We showed that the reconfiguration latency can be minimized and the system works at its highest attainable speed as in direct connections if communication paths can be reconfigured while the processors are operating on data in their local memories. This can be achieved when the partial reconfiguration totally overlaps with the computation time ($T_{comp} \geq T_{reconf}$). Although NoC and NoC-like approaches are scalable enough, they might require huge area and power consumption. Our proposed method is a good candidate among other on-chip communication architectures, if the traffic of the system is known in advance and the number of possible traffic patterns are limited.

There are some disadvantages of NoC architectures such as, large occupied area, not being fast enough when the amount of data flow between communicating modules

is getting larger, data synchronization problem, not having support for single source to multiple sinks. Though their huge area and power consumption, NoC and NoC-like architectures are still good solutions to on-chip communication architecture problems on MPSoC architectures. If one wants to implement a NoC, he/she doesn't need to think over physical placement details, to consider about hard bus macros and to have a deep knowledge of configuration and reconfiguration process. Therefore, in general, embedded system designers find it easier to use NoCs than reconfigurable communications infrastructure. However, we proved that DRP2P architecture outweighs NoC architecture in all three performance metrics such as time, area and power.

To reconfigure interconnects in DRP2P, there is no need to use an external intelligent agent, there is also no need to use an external interface between controller and the target reconfigurable device. As a result, neither an external controller nor an external wiring between these devices is necessary for c^2PCAP reconfiguration engine, which is responsible for the control flow of reconfiguring interconnects in DRP2P. Therefore, the hardware cost can be reduced. As the supply voltages for driving external devices are higher than the supply voltages for the internal operations of the FPGA, the total power consumption is reduced. By storing compressed partial bitstreams on on-chip memory and controlling reconfiguration flow with an internal agent, c^2PCAP core reduces hardware cost and power consumption simultaneously. Furthermore, with the capability of storing compressed partial bitstreams, different partial bitstreams can be stored on-chip memory at a glance.

13. FUTURE DIRECTIONS

Both PFMAP and PFROUT given in Chapters 9 and 10 respectively, exploit particle filters in order to find solutions. Due to the nature of particle filters, they are able to run parallel and independently from each other. In the scope of the thesis, we exploited C++ OPENMP library to insert these algorithms thread level parallelism. As a future research direction, these algorithms can be modified to run on GPUs, Accelerated Processing Units (APUs) or VPUs to speed up the running time of both algorithms. If this can be achieved, because of the nature of particle filters, they tend to give better results by increasing the iteration and particle numbers.

Both PFMAP and PFROUT algorithms work for static mapping at design time. By monitoring the current traffic of the system, they can be adapted to run at execution time as future research direction.

Although, P2P and NoC communication architectures are examined in the scope of the thesis in detail, shared bus is not taken into consideration because of it tends to be unscalable if the number of cores increase in the system. However, shared bus can also be considered for small and medium size applications. Moreover, shared bus can be used in a part of a heterogeneous communication architecture.

In Chapter 11, we give all details of the dynamic reconfigurable point-to-point interconnects for parallel multi-core architectures. Likewise PFMAP and PFROUT, DRP2P works for traffic patterns which are known in advance. If the given architecture has unexpected incoming or outgoing traffic flows, DRP2P might be adapted to monitor the system. Thus, DRP2P can also be used at execution time. In addition to these, there might be some new research directions in order to reduce the reconfiguration time. Configuration clock speed of current reconfiguration engines can be increased if they are placed close to the reconfiguration memory interface. Thus, the reconfiguration engines can be overclocked in order to speed up reconfiguration process.

Processes, such as communication scenario extraction and selection of most appropriate communication infrastructure are being done manually for DRP2P at the moment. For these processes, an automation tool can be designed, which extracts the communication scenario and decides the most suitable interconnection network at a time for a given data intensive application.

The outcomes of both PFMAP (see Chapter 9) and PFROUT (see Chapter 10) can be applied to DRP2P; as the improper location of communicating nodes might increase the size of reconfigurable area(s), which results in the increase both reconfiguration time and on-chip storage directly, a smart mapping like in PFMAP can be utilized for DRP2P. Similarly, inefficient manual routing of wires in DRP2P might increase the size of reconfigurable area(s). Hence, this process also can be done automatically by using routing approach like in PFROUT.

APPENDIX A: USER MANUAL FOR RECONFIGURATION ENGINES AND DRP2P

PCAP, cPCAP and c^2PCAP reconfiguration engines are used to accomplish Dynamic Partial Self-Reconfiguration on Xilinx FPGAs. The cPCAP is the first and the c^2PCAP is the second level extended version of PCAP, where the little letter 'c' stands for "compression". Since c^2PCAP core consists of all properties of both PCAP and cPCAP cores, we only focus on the usage of c^2PCAP core in this part of the thesis. As DRP2P uses c^2PCAP core, it is enough to follow design flow given in Figure 11.7 in Section 11.2. Here, the user must take into account the occupied reconfigurable area, which consists of interconnects between communicating nodes. As the partial bitstream size is proportional to the reconfigurable area, the occupied area for reconfiguration must be kept as small as possible. To minimize reconfigurable area for DRP2P, we applied WelshPowell heuristic graph coloring algorithm [106] to the input task graph. This algorithm gives the same colors to the non-adjacent nodes in a given graph. The physical location of cores are decided by this algorithm. This issue is examined in our DRP2P publication [19] in detail.

The reconfiguration process is achieved through PCAP, cPCAP and c^2PCAP reconfiguration engines within the FPGA instead of using an embedded processor. Since some of Xilinx FPGAs (e.g. pure Spartan-3) do not have an internal configuration access port (ICAP), partial self-reconfiguration on them is done by adding 11 external single wires to it's SelectMAP parallel port.

The second level extended version of PCAP (c^2PCAP) also works on the devices with ICAP module (e.g Virtex-4, Spartan-6 FPGA). For such devices, there is no need to use an external loop to SelectMAP to do self-reconfiguration process. Instead of SelectMAP port, the ICAP module is used as the reconfiguration interface on these devices.

A.1. Configuration Flow for Reconfiguration Engines

The configuration flow of an FPGA for run-time reconfiguration via c^2PCAP is shown in Figure A.1. Initially, target design including c^2PCAP core with empty BlockRAM contents is generated. This generated bitstream called initial configuration. After that by making changes in the design, partial bitstreams are generated for each of different configuration. After obtaining partial bitstream of each design, difference of each partial bitstream is taken with the reference partial bistream. Here we obtain composite bistreams such as p_{12}, p_{13} and etc. Then, reference partial bitstream and joint bitstreams are compressed by using zero run-length coding. They are also converted into the BlockRAM initial files as well. Empty BlockRAM contents in initial design are filled with those BlockRAM files. After filling BlockRAM with these bitstreams, modified initial design is regenerated. Finally, complete bitstream is generated with modified configuration and this full bitstream is downloaded onto the target FPGA through JTAG interface via a host (e.g. PC).

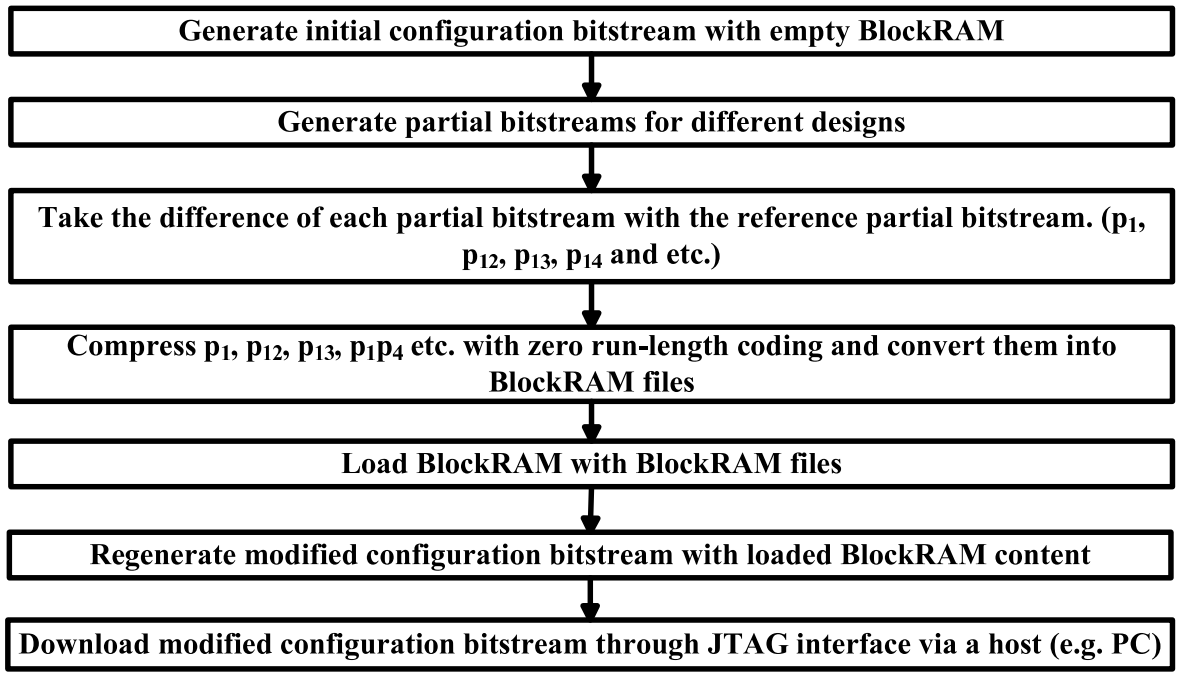


Figure A.1. Configuration flow.

A.1.1. Dynamic Partial Self-Reconfiguration Flow

Each reconfiguration engine behaves as if it is a mirror of SelectMAP port, as same in ICAP. The configuration control flow of our reconfiguration engines is very similar to "SelectMAP configuration Flow Diagram" in [10] except that PROG, INIT, DONE pins are not taken into account in our study. This control flow is illustrated in detail in Figure A.2.

It is experimented, that our latest reconfiguration engine, c^2PCAP core, can run safely at all frequencies up to 75Mhz for Spartan-3 FPGA and 100Mhz for both Virtex-4 and Spartan-6 FPGAs.

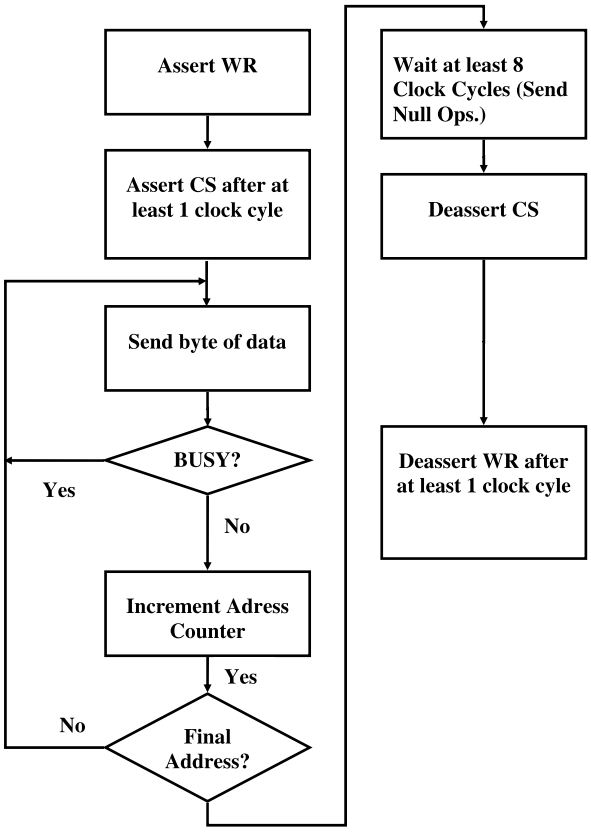


Figure A.2. c^2PCAP core configuration control flow diagram.

A.2. Partial Bitstream Manipulation

Our proposed c^2PCAP is designed for decompressing the compressed bitstreams stored in the BlockRAM, i.e. on-chip RAM of the Xilinx FPGAs. At first, changes

in the design have to be determined. Then for each change in the design, partial bitstreams have to be generated. The following subsections explain how the partial bitstreams are compressed by software at design time and how they are decompressed by hardware at run-time, i.e. c^2PCAP core on the FPGA.

A.2.1. Compression

Compression process and generation of composite bistreams are done at design-time after generating partial bistreams for the target design on the FPGA.

A.2.1.1. Step 1: Partial Bitstream Extraction. Assume that there are two original partial bitstreams p_1 and p_2 for two different reconfigurations in the reconfigurable area and p_1 configures the FPGA before p_2. Let $b_{1,i}$ denote the ith bit value in p_1. Disregarding the initial header and end portions of the bitstreams, each entry in both partial bitstreams can be interpreted as follows:

- If $b_{1,i}$ and $b_{2,i}$ have the same value, then no change is required on the FPGA.
- If $b_{1,i}$ and $b_{2,i}$ have opposite values, then either a wire portion has to be placed or removed on the FPGA.

This interpretation shows that if we can supress the similar and/or opposite portions in the partial bitstreams, we can obtain smaller partial bitstreams. A very simple and straightforward solution is the XOR function:

$$p_{21} = p_{12} = p_1 \oplus p_2 \tag{A.1}$$

where p_{12} can be named as the composite partial bitstream of p_1 and p_2. Note that p_{12} contains the bit values that point out common and different parts in p_1 and p_2

In a dynamically reconfigurable design, there are multiple partial bitstreams to implement different functionalities interchangeably. These functionalities must be de-

fined during the application development time, i.e. at design time. Let N be the number of different functionalities. This means that there must exist N partial bitstreams. Then we can obtain $C(N, 2)$ different composite partial bitstreams by using XOR operation. Figure A.3 shows all possible composite bitstreams when $N = 4$.

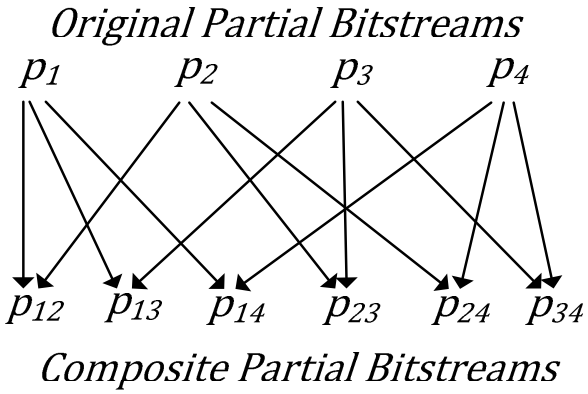


Figure A.3. Possible composite partial bitstreams, N=4.

A.2.1.2. Step 2: Partial Bitstream Compression. After taking the difference of bitstreams, the zero run-length encoding compression technique is applied to original and composite partial bitstreams.

The algorithm checks whether a byte in the partial bitstreams is zero or not. If it is not zero, it is directly written to the BlockRAM. If it is zero, then the algorithm counts the number of successive zero bytes in the partial bitstream and the byte count is written to the BlockRAM by setting the parity bit of the related memory location. Therefore the parity bit of BlockRAM is used as a flag to identify whether the corresponding byte represents the number of successive zero bytes or the data in the partial bitstream. The following example explains our compression method.

On the left side of the Figure A.4 we can see a portion of the uncompressed partial bitstream, on the right side the compressed partial bitstream as it is stored in the BlockRAM. In uncompressed partial bitstream the bytes are ordered from left to right whereas the ordering is from right to left in the compressed bitstream file. Since the first four bytes are different from zero (in this case “20”, “01”, “04”, “02”) they are stored directly in the uncompressed bitstream. The fifth and sixth bytes are zero and

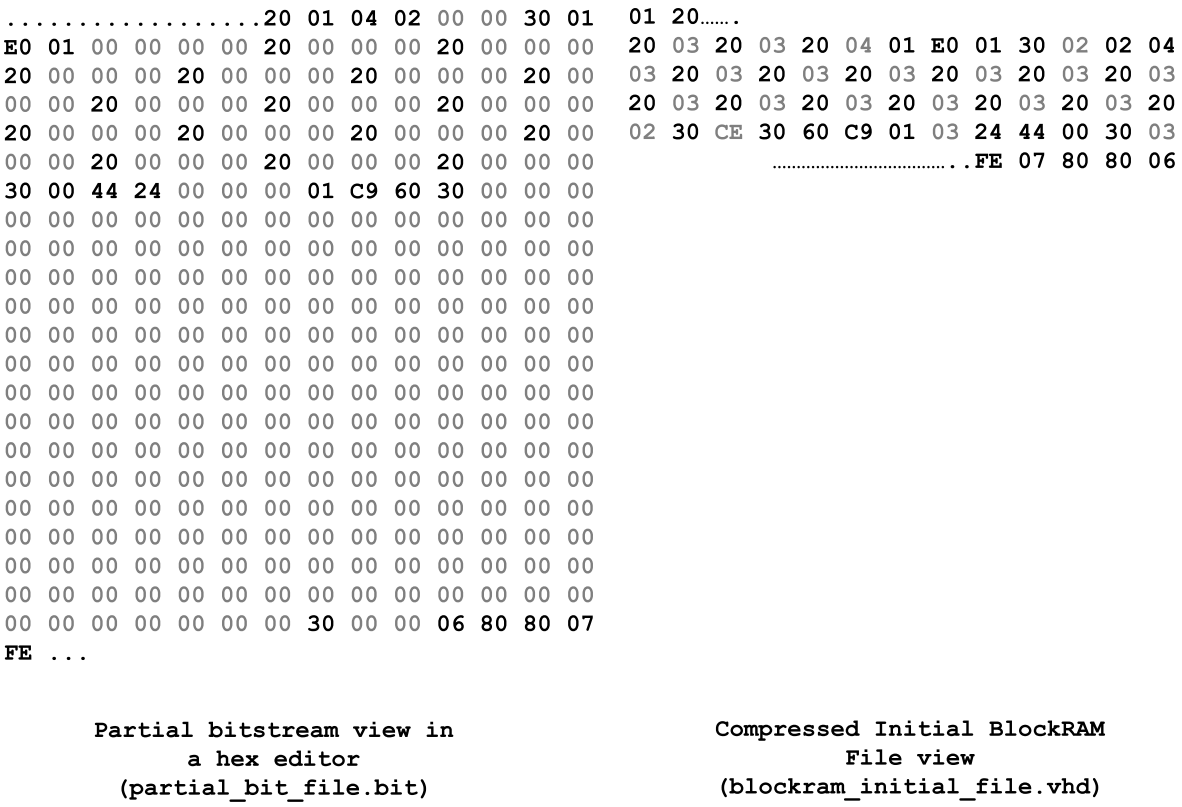


Figure A.4. Bitstream compression example with zero run-length encoding.

there is no more zero bytes consecutive, so instead of storing these two sequential zero bytes, the byte "02", which is actually the number of zeros, is stored. For example, after the "01 C9 60 30" byte sequence, there are 206 zero bytes. Instead of storing 206 bytes, we simply store one byte the "CE", which is the hexadecimal value of 206. If the number of consecutive zeros are more than 255, they are in multiple bytes. For example, assume there are 1056 consecutive bytes in the bitstream, then they are represented as "24FFFFFFFF" byte sequence.

A.2.1.3. Step 3: Partial Bitstream Picking. After compressing the original bitstreams and their differences (XORed pairs), the smallest suitable partial bitstream set must be elected to be stored on the on-chip memory. In Section 7.2.3, we have already given all details in order to select optimal compressed bitstream set to be stored on the on-chip memory. Hence, this part will not be discussed in this part of the manual.

A.2.2. Decompression

The decompression process is done at run-time by the reconfiguration engine. When a reconfiguration request comes to the system, the reconfiguration engine starts to read partial bitstreams from BlockRAMs. For each byte stored on BlockRAM, our reconfiguration engine controls whether this is an original byte or a number representing the consecutive "00" bytes. Each byte read from BlockRAM is sent to the configuration interface such as SelectMAP or ICAP. In the following subsections, decompression process is examined in detail.

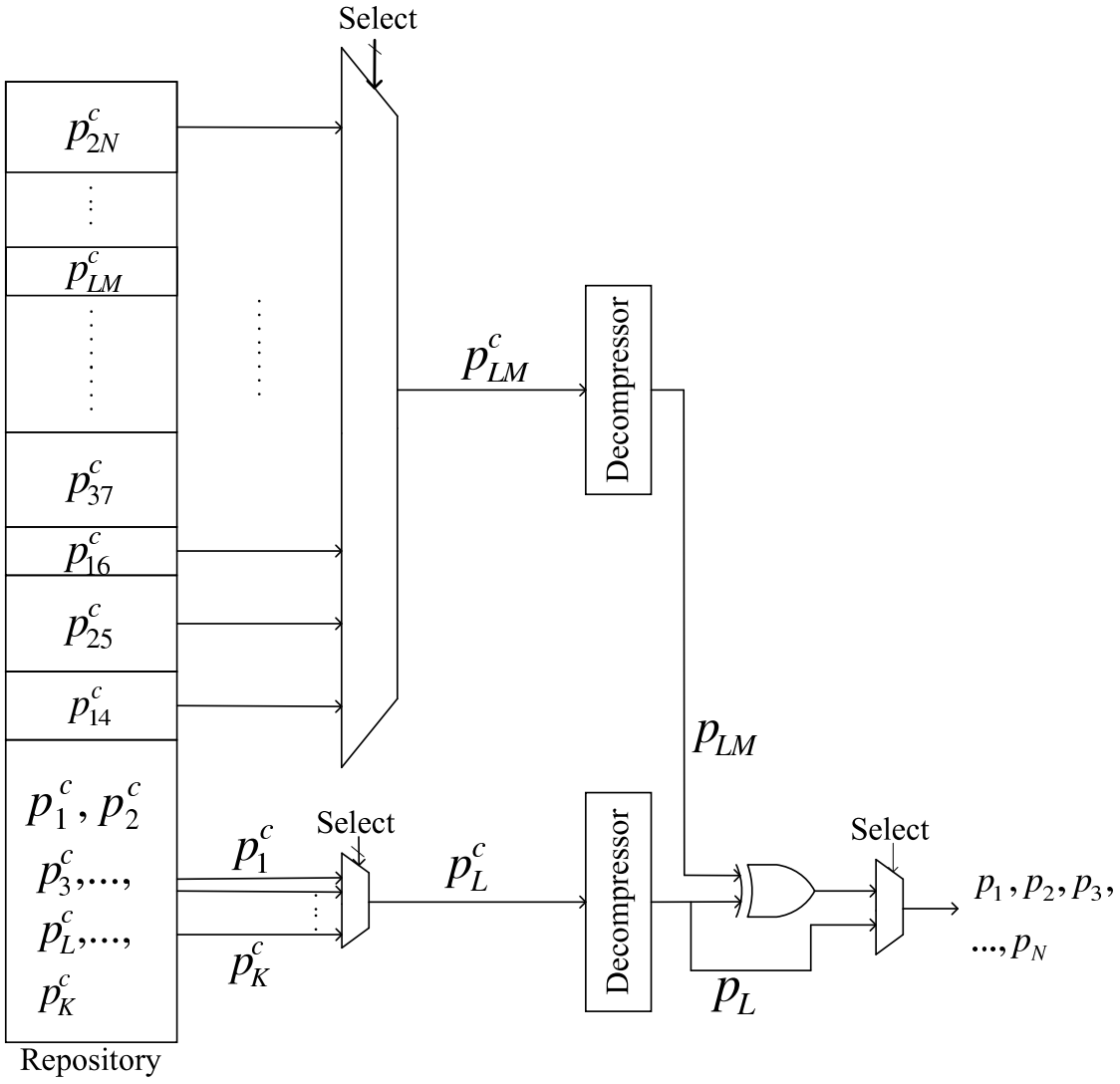


Figure A.5. Decompressing and extracting original bitstreams in two steps (PR: Any reference bitstream, N: # of original bitstreams, M<=N, K<=N, L<=K).

A.2.2.1. Step 1: Partial Bitstream Decompression. The decompression is straightforward as shown in Figure A.5. The identification of whether a byte in compressed bitstream is controlled by the parity bit of appropriate byte. If the parity bit of the byte value in BlockRAM file is set "1", then it means that the byte shows the number of zero bytes and as many zeroes are generated as the number states. Otherwise the byte value is automatically written to the output.

A.2.2.2. Step 2: Partial Bitstream Extraction. If one of original bitstreams has been selected as a desired design at that time, it is directly downloaded to the target device. Otherwise, the XOR operation is applied to p_r and p_{rm}, where p_r is one of original bitstreams and p_{rm} is a composite bitstream. The operation $p_r \oplus p_{rm}$ is equivalent to $p_r \oplus (p_r \oplus p_m)$, which gives p_m.

In the following subsections, configuration interfaces SelectMAP and ICAP will be examined in detail.

A.2.3. SelectMAP Specific Topics

Table A.1. SelectMAP port pin descriptions.

Signal Name	**Direction**	**Description**
CCLK	Input	Configuration clock
M[2:0]	Input	Configuration Mode selection
D[7:0]	Input	Byte-wide configuration data input
$\overline{\textbf{CS}}$	Input	Active Low Chip Select input
$\overline{\textbf{WRITE}}$	Input	Active Low Write Select input
BUSY	Output	Handshaking signal to indicate successful data transfer

SelectMAP configuration data is loaded one byte at a time presented on the D[0:7] bus on each rising CCLK edge. Two extra control signals are present for SelectMAP, $\overline{CS}$ and $\overline{WRITE}$. These signals must both be asserted Low for a configuration byte to be transferred to the FPGA [10]. In addition, Table A.1 shows the SelectMAP

pin descriptions. All signals, including 'BUSY' indicator signal, are used for c^2PCAP hardware connection.

Table A.2. Loopback hardware pin connections for SelectMAP port on Spartan-3 starter kit board.

Signal Name	**FPGA Output Pins**	**Connector Type and Pin Nr**	**FPGA Input Pins**
CCLK	"B6"	A2,pin 21 --→ B1,pin 39	FPGA CCLK "T15"
D[7:0]	"B5-B4-D10-D8"	A2,pin 19,17,15,13,11,9,7,5	"M11-N11-P10-R10"
	"D7-E7-D6-D5"	B1,pin 40,7,9,11,13,15,17,19	"T7-R7-N6-M6"
$\overline{\text{CS}}$	"A7"	A2,pin 23 --→ B1,pin 20	FPGA $\overline{CS\_B}$ config "R3"
$\overline{\text{WRITE}}$	"A8"	A2,pin 25 --→ B1,pin 5	FPGA $\overline{RD\_WR\_B}$ config "T3"
BUSY	"P9"	Handshaking signal to indicate successful data transfer	Any input pin

Table A.2 shows the pinout descriptions for loopback hardware to the SelectMAP port. In the second column of this table, there are FPGA output pins for SelectMAP port, which are defined in user constraint file(ucf) of design as outputs of FPGA. These all output pins are on A2 Expansion Connector of SPARTAN-3 Starter Kit Board and they are directly connected to the FPGA Input pins with 11 external single wires. The FPGA Input pins are on the most right column of Table A.2 and they are physically on B1 Expansion Connector of SPARTAN-3 Starter Kit Board. The Spartan-3 Starter Kit board has three 40-pin expansion connectors labeled A1, A2, and B1. The A1 and A2 connectors are on the top edge of the board as in Figure A.6. Connector A1 is on the top left, and A2 is on the top right. The B1 connector is along the right edge of the board as in Figure A.6. For more details of expansion connector features please refer to [10].

Notice that the most significant bit (MSB) of each configuration byte is on the D0 pin. Figure A.7 shows two bytes (0xABCD) being reserved.

In order to achieve byte swapping, D[7] output ("B5", pin 19 on A2) is connected to D[0] input ("M11", pin 40 on B1) and D[6] output ("B4", pin 17 on A2) is connected to D[1] input ("N11", pin 7 on B1) and so on. It is shown in detail in Table A.3.

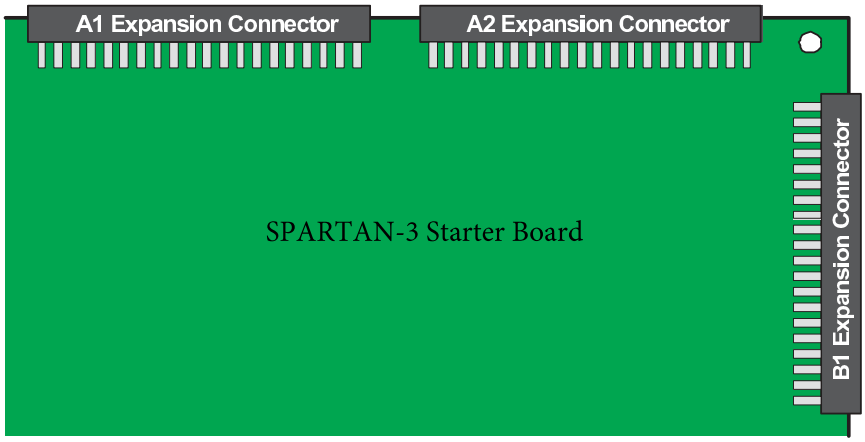


Figure A.6. Spartan-3 starter kit board expansion connectors [10].

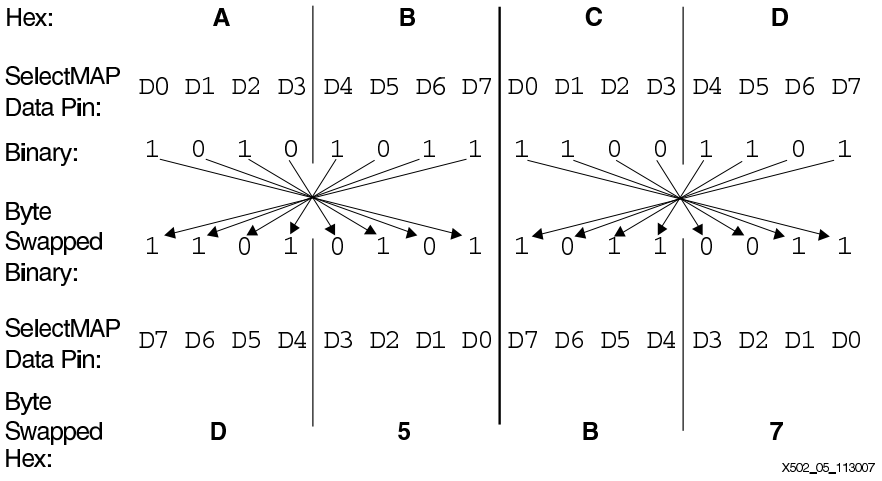


Figure A.7. Byte-Swapping Example [10].

The reserved pins 7, 9, 11, 13, 15, 17, and 19 (D[1-7]) on B1 connector provide the signals required to configure the FPGA in Master or Slave Parallel mode. In addition, pin 40 on B1 is used either as DIN for serial configuration or as D[0] for parallel configuration. Since, this design uses Parallel SelectMAP port, the pin 40 on B1 is used as D[0] input.

For the lower frequency values ($< 75MHz$ for Spartan-3 and $< 100MHz$ for Virtex-4 and Spartan-6), the CCLK signal can be chosen CLKFX again by changing the value of this output. The CLKFX output of a DCM is a multiplied frequency of the CLKIN frequency by the attribute-value ratio (CLKFX_MULTIPLY/CLKFX_DIVIDE) to generate a clock signal with a new target frequency. Here CLKFX_MULTIPLY is the frequency multiplier constant which is an integer from 2 to 32, and in the same way CLKFX_DIVIDE is the frequency divisor constant which is an integer from 1 to 32 [10].

Table A.3. Loopback hardware byte swapping for SelectMAP port on Spartan-3 starter kit board.

FPGA Output Pins	**FPGA Input Pins**
D[7], pin 19 on A2	D[0] pin 40 on B1*
D[6], pin 17 on A2	D[1] pin 7 on B1
D[5], pin 15 on A2	D[2] pin 9 on B1
D[4], pin 13 on A2	D[3] pin 11 on B1
D[3], pin 11 on A2	D[4] pin 13 on B1
D[2], pin 9 on A2	D[5] pin 15 on B1
D[1], pin 7 on A2	D[6] pin 17 on B1
D[0], pin 5 on A2	D[7] pin 19 on B1

As mentioned above, the extra control signals $\overline{CS}$ and $\overline{WE}$ must both be asserted Low for a configuration byte to be transferred to the FPGA. Furthermore, write select input signal($\overline{WE}$) must be asserted at least one clock cycle after asserting the Chip Select input($\overline{CS}$) signal. In the same way, write select input signal($\overline{WE}$) must be deasserted at least one clock cycle before deasserting the Chip Select input($\overline{CS}$) signal

[10] as in Figure A.8.

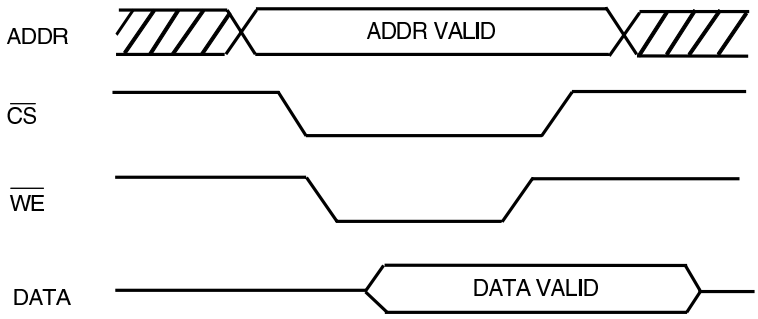


Figure A.8. Write cycle timing diagram [10].

A.2.4. Internal Configuration Access Port (ICAP) Specific Topics

The Internal Configuration Access Port (ICAP) allows access to configuration data in the same manner as SelectMAP. ICAP has the same interface signaling as SelectMAP other than the data bus, which is separated into read and write data buses. ICAP has a chipselect signal (CS), a read-write control signal (RD), a clock (CLK), a write data bus (DIN), and a read data bus (OUT). ICAP can be configured to two different data bus widths, 8 bits or 32 bits. When the 8-bit ICAP interface is used, the data is byte-reversed like SelectMAP. When the 32-bit interface is used, the data is not reversed, which is the same as SelectMAP32. Since it is an internal component, for the self reconfiguration of the target device, there is no need to use external wiring to access this port [10].

The ICAP interface can be used to perform readback operations or partial reconfiguration. When using ICAP for partial reconfiguration, the user must avoid changing the logic or interconnect which the ICAP is itself connected to. ICAP can also be used to read or write to the configuration registers(e.g. STAT, CTL, or FAR registers) [10].

There are two ICAP sites in Virtex-4 devices: TOP and BOTTOM. The implementation has the two interfaces share the same underlying logic. The only difference between them is their location on the chip and the interconnect to which they can be connected. The two interfaces can never be active at the same time. The default site for a single ICAP is the TOP site, because the TOP site is active after configuration

by default. If both sites are used, the TOP site must be activated first before switching to the BOTTOM site [10].

Similarly, the partial self-reconfiguration on a Spartan-6 FPGA can be achieved through ICAP module of the device up to 100MHz with a 16-bit interface only. However, Spartan-6 FPGAs do not support 8-bit and 32-bit for ICAP. Therefore, the maximum self-reconfiguration speed that can be achieved on a Spartan-6 FPGA is 200MB/s. In addition to this, the external SelectMAP configuration interface can also be used for initial or partial reconfiguration. Note that some of Spartan-6 FPGAs (e.g. XC6SLX4 devices or devices using TQG144 or CPG196 packages.) do not offer the SelectMAP interface. Likewise Virtex-4 devices, Spartan-6 FPGAs offer the partial reconfiguration in a two dimensional fashion [10]. The configuration details for Spartan-6 FPGA can be found in Section 6.1.

A.2.5. Slice Based Bus Macros

For a lossless data communication between reconfigurable and fixed area in a reconfigurable design, the designer must use a type of bus macro, which guarantees a safe data flow in both directions at the time of reconfiguration process. Such a lossless communication can be implemented by tri-state buffers offered by Xilinx Inc. However, some FPGA series from Xilinx such as Virtex-4, Spartan-6 and pure Spartan-3 families have no tri-sate buffers. Hence, for these device families a new type slice-based bus-macro, which acts as a tri-state buffer, should be developed.

As in Figure A.9, independent from the data flow direction, the slices in the reconfigurable area are all identity functions. For the communication from fixed area to reconfigurable area each slice in the fixed area is implemented as an identity function, however for the reverse communication transparent latches with control signals are implemented. If the control signal is set to '0', it means that the data flow from reconfigurable area to fixed area is allowed, otherwise it is prohibited to protect the fixed part of the design from undesired transient signal switchings coming from reconfigurable part at the time of reconfiguration.

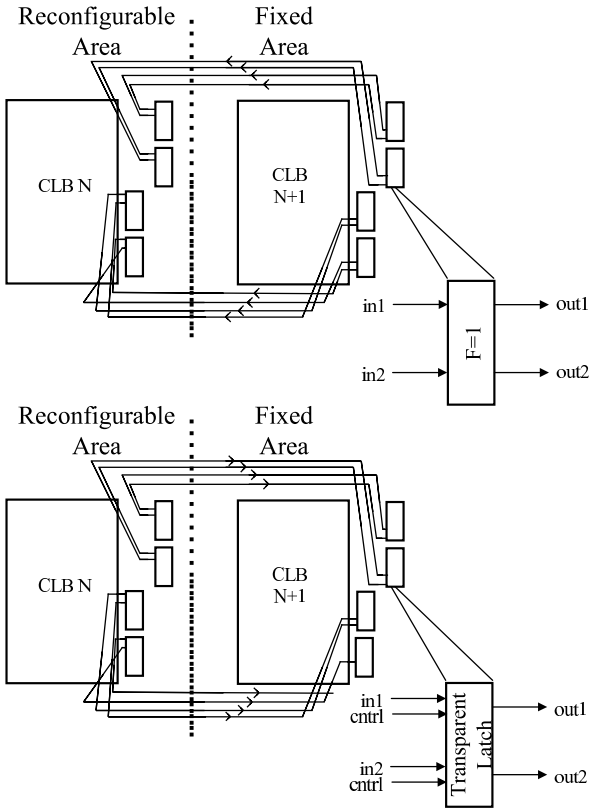


Figure A.9. General structure of slice based bus macro.

Contrary to and-gate implementation, transparent latches keep the last data before reconfiguration starts, thus there is no data loss during reconfiguration. Note that for the implementation of slice based bus macros, the CLBs need not to be consecutive. According to requirements of the design, a new CLB column or multiple CLB columns can take place within the area of bus macros.

These two types of slice based bus macros are used, if the reconfigurable area is on the left side. For the reverse type designs, there must be two other bus macros available. By swapping the CLB blocks the two additional bus macros are derived from the bus macros in Figure A.9.

There are no slice based HBMs offered by Xilinx for Spartan-6 FPGAs. However, there are some bus macros which are directly used for some target FPGA families (e.g. Virtex-5, Virtex-6) from Xilinx. Instead of designing a new bus macro we have manipulated and adapted Virtex-5 bus macros to the Spartan-6 bus macros. The 4-bit-width single slice HBM component is illustrated in Figure A.10. In Figure A.11,

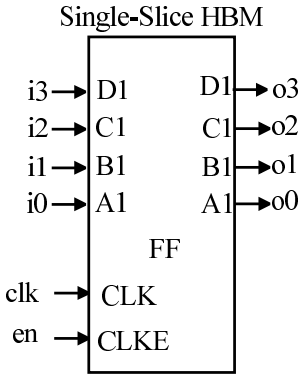


Figure A.10. 4-bit single slice Spartan-6 HBM.

the internal structure and connections of our single slice synchronous Spartan-6 bus macro can be seen.

A.3. The Configuration Architecture of Virtex-4 and Pure Spartan-III FPGAs from XILINX

As shown in Figure A.12, Virtex-4 configuration memory is arranged in frames, which are the smallest addressable segments that are tiled about the device. In contrast to previous generation of Virtex families, Virtex-4 architecture is composed of fixed-length frames, each consisting of 41 words (each one 32-bit) [10]. Each configuration frame has a unique 32bit address that can be divided into Block type, Top/Bottom indicator, Row address, Column address, Minor address. Virtex-4 frames are columns that span 8 CLB (Control Logic Block)+1 HCLK(Horizontal Clock)+ 8 CLB or the equivalent logic such as, 2 BlockRAM + 1 HCLK + 2 BlockRAM or 16 IOB (Input/Output Block) + 1 HCLK + 16 IOB. Some special logics such as JTAG, ICAP etc. are not taken into account by the columns [10].

For the Virtex-4 device, the Top/Bottom indicator specifies whether the target is at the top or at the bottom half of the chip. The Block type indicates the type of a block, if it is “00” then a block of CLB, IOB, DSP or GCLK, if it is “01” the BlockRAM interconnect, if it is “10” the BlockRAM content is selected. The Row address (HCLK row) specifies which row of the target device is selected. The Column address selects a major column, such as a column of CLBs. Column addresses start at 0 on the left

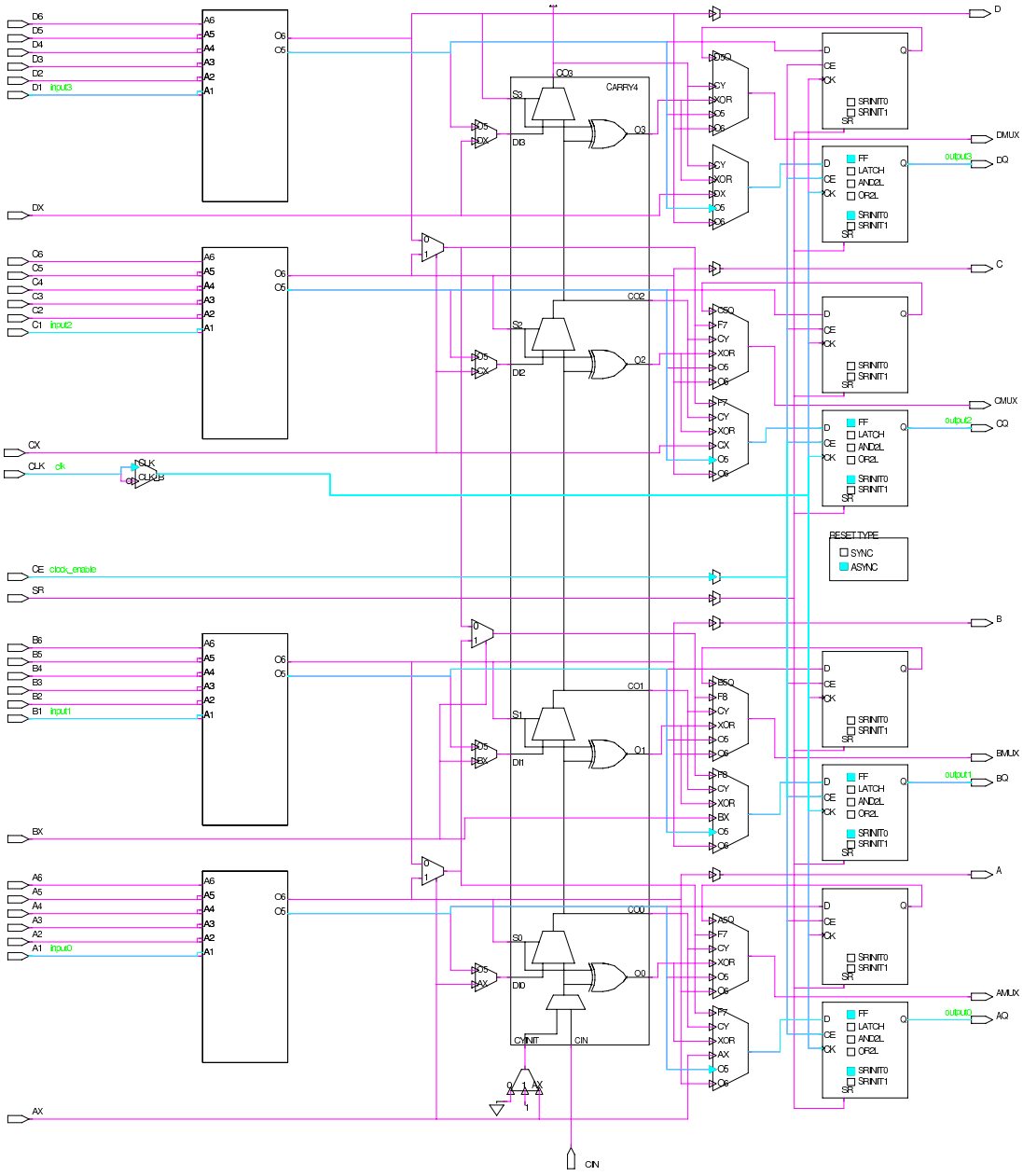


Figure A.11. Inside of Spartan-6 slice based HBM.

and increase to the right. The Minor address selects a memory-cell address line within a major column.

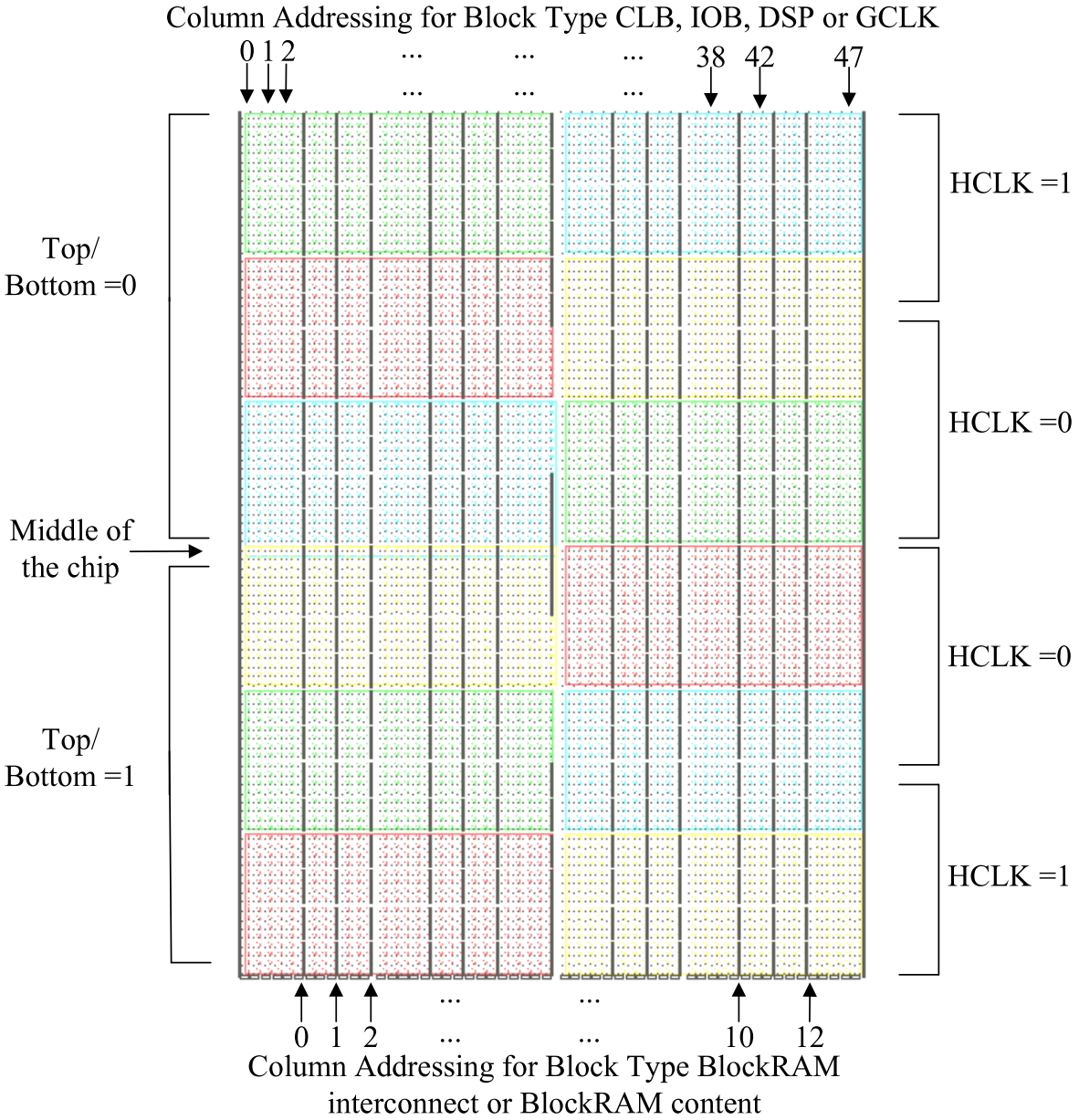


Figure A.12. Virtex-4 configuration frame addressing scheme [11].

The Spartan-3 FPGA configuration memory can be visualized as a rectangular array of bits. The bits are grouped into vertical frames that are one-bit wide and extend from the top of the array to the bottom. A frame is the atomic unit of configuration. It is the smallest portion of the configuration memory that can be written to or read from [10]. Frames are grouped into larger units called columns. Spartan-3 devices have different types of columns: TERM(L/R), IOI (L/R), CLB, BlockRAM content, BlockRAM Interconnect, GCLK columns.

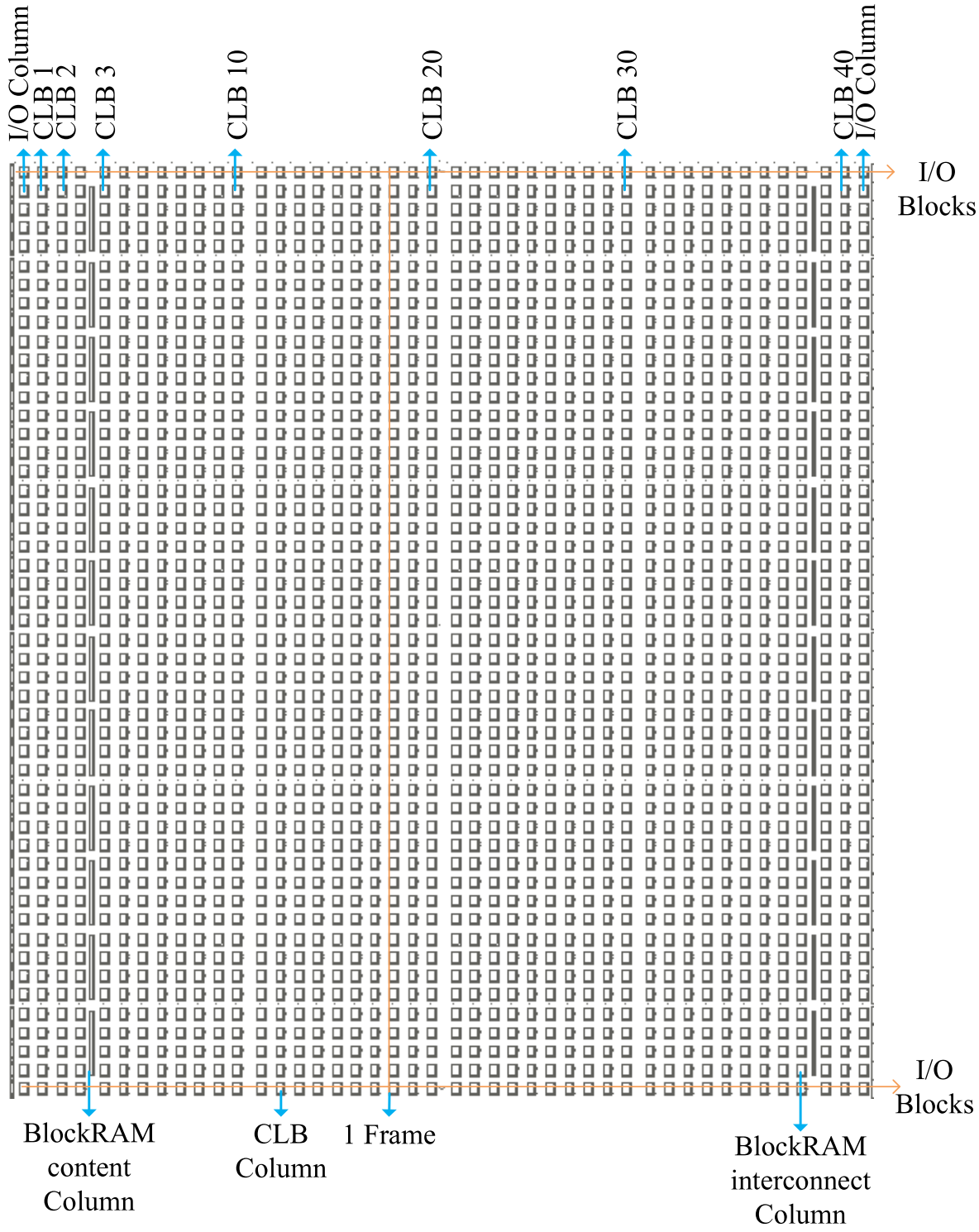


Figure A.13. Configuration column addressing scheme for Spartan-3S1000 FPGA.

Certain IOBs that are located at the top and bottom of the device are configured in a CLB column, along with the CLBs. Since IOBs that are located at the top and bottom of the device is also a part of the appropriate CLB column, the designer should use no pin-outs while choosing a reconfigurable area in a pure Spartan-3 device. In addition to this, Spartan-3 doesn't support glitch-less reconfiguration; that is, unmodified bits in a partially reconfigured column in a Spartan-3 device are temporarily reset during the reconfiguration process. Therefore, if this method is used, designers make sure to manage these glitches by using handshaking for design communication. In Figure A.13, FPGA Editor view of a Spartan3s1000 and column addressing of this device can be seen. In FPGA Editor, the TERM(L/R) and GCLK columns are not visible to the designer.

For the Spartan-3 devices three different address pieces specify a frames address: the Column address, the Major address, and the Minor address. The Column address indicates what kind of data is being loaded ("00" for TERM, IOI, CLB and GCLK, "01" for BlockRAM, "10" for BlockRAM Interconnect columns). The Major address, indicates where (vertically) in the device the frame lies. The Minor address, indicates where within the Major address the frame lies [10].

In this work, for the partial reconfiguration of the Spartan-3 devices, the Bitgen "PartialMask" flow is used. This feature allows users to pick which configuration columns are included in an active reconfiguration bitstream. According to "System Reference Development Guide of XILINX" there are six types of PartialMask settings: PartialGCLK, PartialLeft, PartialRight, PartialMask0 <mask>, PartialMask1 <mask> and PartialMask2 <mask> [10]. With PartialMask0, the columns in the column address "00" as indicated by the hexadecimal mask value are added to partial bitstream. In the same way, with PartialMask1, the columns in the column address "01" and with PartialMask2, the columns in the column address "10" are added to partial bitstream. With PartialGCLK, the center global clock column is added to the list of columns written to a partial bitstream. In PartialLeft/PartialRight, as the name implies, the left/right half of the device (excluding the global clock column) is added to list of columns written to a partial bitstream.

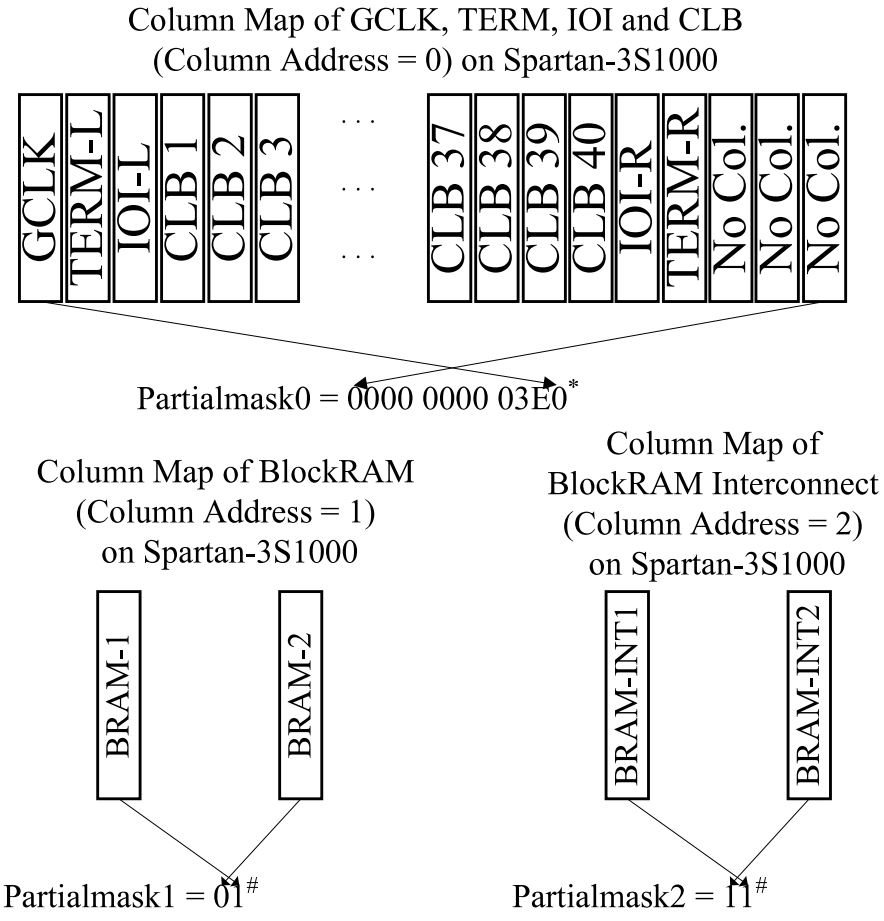


Figure A.14. Partial mask derivation example for Spartan-3S1000 FPGA.

A PartialMask derivation example is shown in Figure A.14. This figure is similar to "PartialMask <mask> Derivation for Example 1" in [10]. However, in [10] only the Virtex-II device is targeted. Unfortunately there is no detailed information for the column map of Spartan-3 devices. Since the Spartan-3 family is also based on Virtex-II architecture, the column map and PartialMask values of Spartan-3 family is derived from "PartialMask <mask> Derivation for Example 1" in [10].

The <mask> is a hexadecimal string that indicates which columns within a memory block are included in the partial bitstream. In PartialMask each binary '1'indicates a column that is included in the partial bitstream, while each binary '0' indicates a column that is excluded from the partial bitstream [10]. The value of <mask> can be given in hexadecimal, binary or decimal. For the sake of clarity, in this illustration, the <mask> value of PartialMask0 is in hexadecimal, the <mask> values of PartialMask1 and PartialMask2 are shown in binary. With PartialMask0 = "0000000003E0$_h$" from the 3. CLB column to 7.CLB column, totally 5 CLB columns are added to the list of columns written to the partial bitstream. In the same way, with PartialMask1 = '01$_b$' the BlockRAM at the right side and with PartialMask2 = '11$_b$' the right and left BlockRAM-Interconnect columns are included in the partial bitstream. Note that the Spartan-3S1000 has only 2 BlockRAM and 2 BlockRAM Interconnect columns.

A.3.1. Creating Partial Reconfiguration Bitstreams

Partial reconfiguration of Spartan-3 devices can be accomplished through the parallel slave SelectMAP or JTAG interfaces. In addition to these configuration interfaces, the ICAP interface can be used for Virtex-4 devices. Instead of resetting the device and performing a complete reconfiguration, new data is loaded to reconfigure a specific area of a device, while the rest of the device is still in operation [10].

A.3.1.1. Difference-Based Partial Reconfiguration:. The ***-g ActiveReconfig:Yes*** switch is required for active partial reconfiguration, meaning that the device remains in full operation while the new partial bitstream is being downloaded. If ***ActiveReconfig:Yes***

is not specified (or ***-g ActiveReconfig:No*** is specified), then the partial bitstream contains the Shutdown and AGHIGH commands used to deassert DONE. Additionally, the ***-g Persist:Yes*** switch is required when utilizing partial reconfiguration through the SelectMAP mode. This switch allows the SelectMAP pins to persist after the device is configured, which allows the SelectMAP interface to be used for partial reconfiguration. The ***-g Persist:Yes*** setting is also required for the initial bitstream. However, for the reconfiguration through ICAP interface, the ***-g Persist:No*** setting is used. This setting is also the same when the ***-g Persist*** is not set. That is the ***-g Persist:No*** setting is the default setting.

A difference-based partial reconfiguration bitstream can be created with the BitGen utility using the ***-r*** switch. This switch produces a bitstream that contains only the differences between the ***input .ncd file*** and ***the original bit file***.

Examples:
Generic Example for SelectMAP interface:

bitgen -g ActiveReconfig:Yes -g Persist:Yes -r < ***original.bit*** > < ***new.ncd*** > < ***new.bit*** >

Generic Example for ICAP interface:

bitgen -g ActiveReconfig:Yes -g Persist:No -r < ***original.bit*** > < ***new.ncd*** > < ***new.bit*** >

Test Example for SelectMAP interface:

bitgen -g ActiveReconfig:Yes -g Persist:Yes -r and_test.bit and_test2.ncd and_test2_partial.bit

Test Example for ICAP interface:

bitgen -g ActiveReconfig:Yes -g Persist:No -r and_test.bit and_test2.ncd and_test2_partial.bit

Create a Partial Bitstream to Restore the Original Design for SelectMAP interface:

bitgen -g ActiveReconfig:Yes -g Persist:Yes -r and_test2.bit and_test.ncd and_test_partial.bit

Create a Partial Bitstream to Restore the Original Design for ICAP interface:

bitgen -g ActiveReconfig:Yes -g Persist:No -r and_test2.bit and_test.ncd and_test_partial.bit

These files produce a configuration file ***(and_test2_partial.bit)*** that only configures

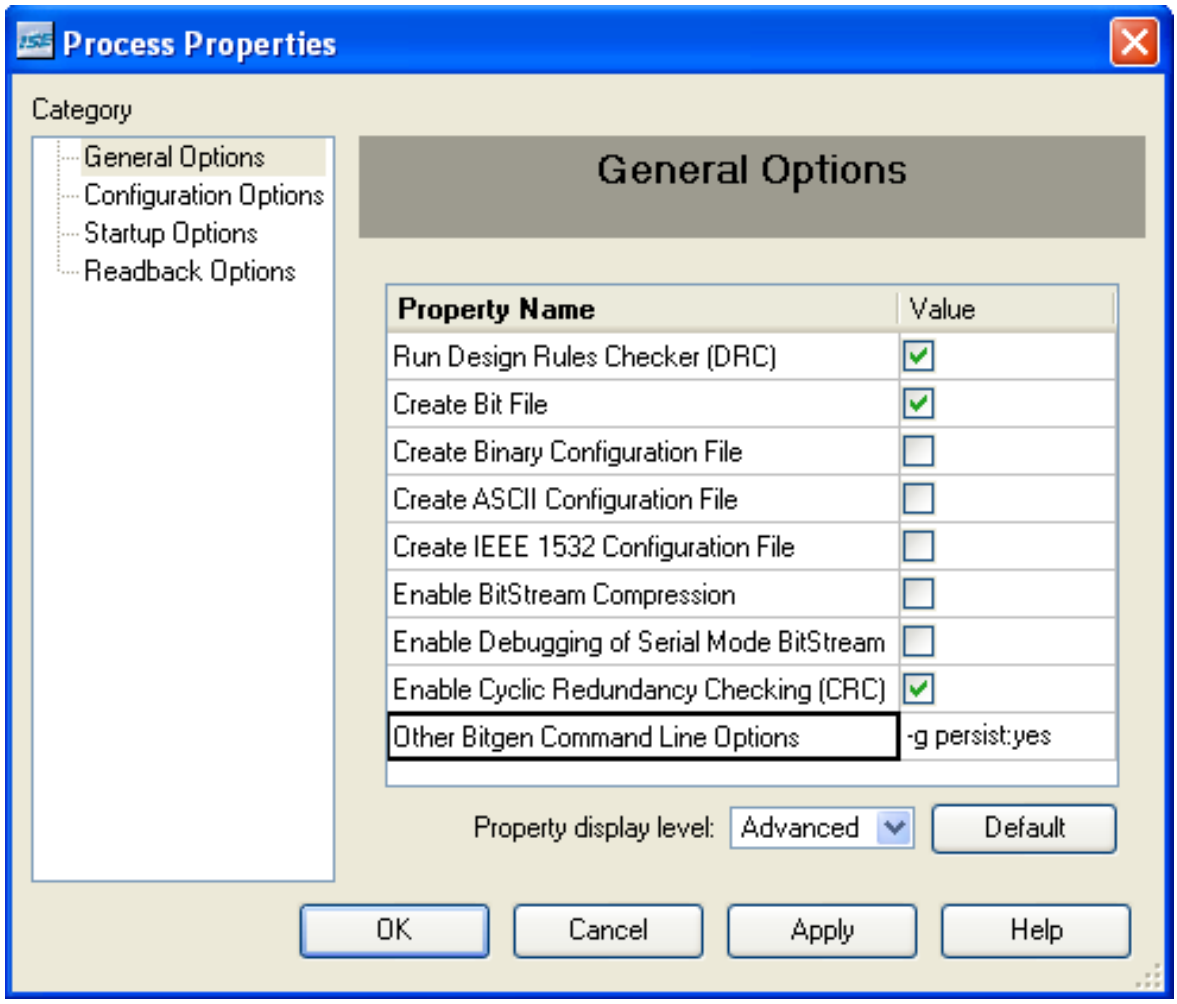


Figure A.15. -g Persist:yes setting for initial bitstream (SelectMAP)

the frames that are different between ***and_test*** *and* ***and_test2***. When downloading this file, the ***and_test*** configuration file MUST already be programmed into the device [10].

Since the ***-g Persist:Yes*** setting must also be required for the initial bitstream, a designer can accomplish this by right-clicking the mouse over the "Generate Programming File" in the "Processes" Window in "Project Navigator" interface of Xilinx ISE (Integrated Synthesis Environment). After right-clicking, user encounters with the "Process Properties" window, there under the "General Options" tab for the property "Other Bitgen Command Line Options" ***-g Persist:Yes*** siwtch can be set as shown in Figure A.15. For the ICAP interface, either ***-g Persist:No*** switch can be set or this field can be left blank (default is set to no). As mentioned previously, in order to be able to use partial reconfiguration through the SelectMAP mode, this step must be done before generating the initial complete bitstream. Here it is noted that, the settings of bitgen tool is case insensitive (e.g. ***-g Persist:Yes*** $\equiv$ ***-g persist:yes***).

A.3.1.2. -g PartialMask Options of BitGen for Partial Reconfiguration:. The alternative is including in the partial bitstream all the frames corresponding to the reconfigurable area, not just the ones that change between two designs. This can be achieved by using the options ***-g PartialMask*** of bitgen. Here it is noted that, the ***-g PartialMask*** options of bitgen is not applicable for Virtex-4 devices.

In addition to ***-g Persist:Yes*** setting, there is one another thing, which must also be done before generating the initial complete bitstream. It is nothing more than defining a Place And Route (PAR) guide design file which is actually used for ***-g PartialMask*** options of BitGen.

User can accomplish this by expanding the "Implement Design" tab, and there right-clicking the mouse over the "Place & Route" and choosing "Properties" in the "Processes" Window in "Project Navigator" interface of Xilinx ISE. After right-clicking, user encounters with the "Place & Route Properties" window, and there, for the prop-

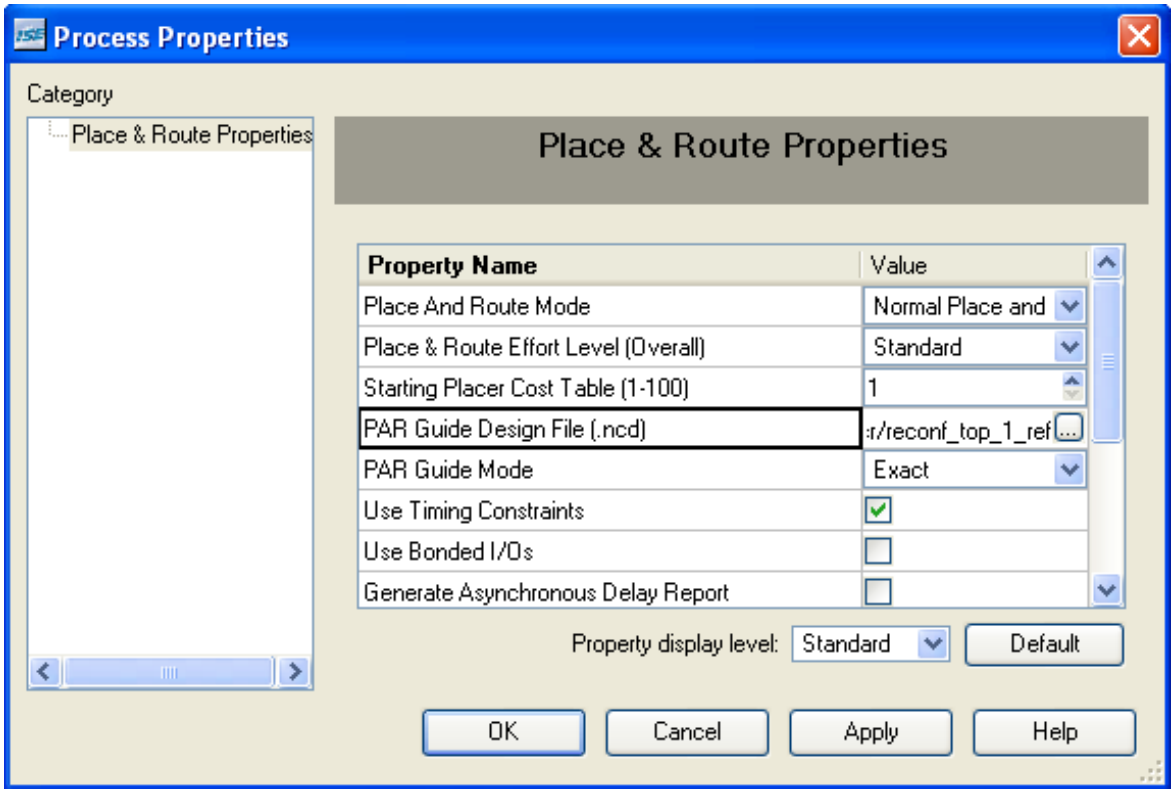


Figure A.16. Defining a PAR guide design file.

erty "(PAR) Guide Design File (.ncd)" user may type the path of your Native Circuit Design (NCD) file as in Figure A.16. Afterwards user must also set the "PAR Guide Mode" to "exact". As mentioned before, in order to be able to use partial reconfiguration with ***-g PartialMask*** options of BitGen, this step must be done before generating the initial complete bitstream.

We use ***-g PartialMask*** options of BitGen to make sure the partial bitstream includes all the frames for the reconfigurable area; that is, the reconfiguration process, which is done by Difference-Based approach, can also be accomplished by ***-g PartialMask*** options of BitGen, so reconfiguration with ***PartialMask flow*** can be used as a verification method.

Since there is no information about ***PartialMask flow*** of BitGen for Spartan-3 FPGAs, it seems that, the Spartan-3 architecture does not support ***PartialMask flow***. Furthermore, there is no information about ***PartialMask flow*** of BitGen for Spartan-3 FPGAs in any Spartan-3 specific documents such as [10]. However,

the information about ***PartialMask flow*** of BitGen for Virtex FPGAs is available in [10] ("Virtex-II Pro and Virtex-II Pro X FPGA User Guide"). Indeed, according to the ***Development System Reference Guide for Xilinx ISE 8.2i*** six different ***PartialMask flows*** of BitGen are applicable for Spartan-3 FPGAs [10]. However, it is not described in detail how ***PartialMask flows*** can be applied for Spartan-3 FPGAs, it is only claimed that Spartan-3 architecture also supports ***PartialMask flow***. Since ***PartialMask flow*** of BitGen for Virtex FPGAs is described in [10] in detail, we can utilize and adapt this information for Spartan-3 FPGAs.

Table A.4. Spartan-3 bitstream column types.

Column Type	**# of Frames per Column**	**# of Columns per Device**	**Column Address**
TERM(L/R)	2	2	00
IOI (L/R)	19	2	00
CLB	19	# CLB columns	00
BRAM	76	# BRAM columns	01
BRAM Interconnect	19	# BRAM columns	10
GCLK	3	1	00

Table A.4 shows Spartan-3 bitstream column types. To be able to achieve ***PartialMask flow*** of BitGen for Spartan-3 FPGAs, we firstly need to have some information about column types and map of Spartan-3 FPGA, thus by giving appropriate column number (column address) we can reconfigure the necessary column(s). These columns are given in Table A.4 in detail.

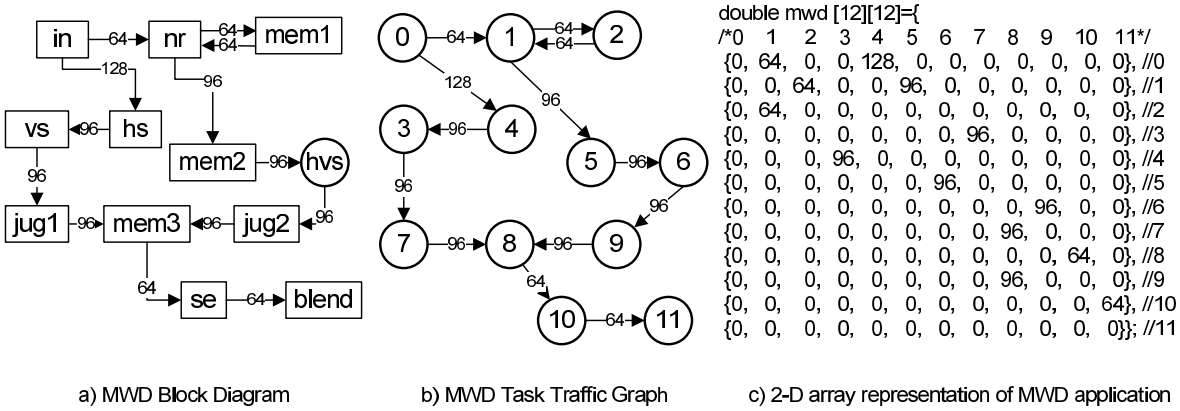
The BitGen "PartialMask" feature allows users to pick which configuration columns are included in an active reconfiguration bitstream. PartialMask bitstreams are intended only for active partial reconfiguration, and must be used with the ***-g activereconfig:yes*** BitGen switch. Bitstreams that are created using this flow cannot be used for initial configuration, since they do not include the START command or allow for the startup sequence.

There are six *PartialMask* settings:

- **PartialGCLK:** Adds the center global clock column to the list of columns written to a partial bitstream. Equivalent to the PartialMask0:1 setting.
- **PartialLeft:** Adds all columns on the left side of the device, excluding the global clock column, to the list of columns written to a partial bitstream.
- **PartialRight:** Adds all columns on the right side of the device, excluding the global clock column, to the list of columns written to a partial bitstream.
- **PartialMask0 <mask>:** Adds columns in BA0 (Block Address 0: GCLK, IOB, IOI, and CLB columns) as indicated by the hexadecimal mask to the list of columns written to a partial bitstream.
- **PartialMask1 <mask>:** Adds columns in BA1 (BRAM columns) as indicated by the hexadecimal mask to the list of columns written to a partial bitstream.
- **PartialMask2 <mask>:** Adds columns in BA2 (BRAM Interconnect columns) as indicated by the hexadecimal mask to the list of columns written to a partial bitstream.

According to the aforementioned *PartialMask* settings, desired columns of a Spartan-3 FPGA can be reconfigured.

APPENDIX B: USER MANUAL FOR PFMAP




```
double mwd [12][12]={
/*0   1   2   3   4   5   6   7   8   9   10  11*/
 {0,  64,  0,  0, 128,  0,  0,  0,  0,  0,  0,  0}, //0
 {0,  0,  64,  0,  0,  96,  0,  0,  0,  0,  0,  0}, //1
 {0,  64,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0}, //2
 {0,  0,  0,  0,  0,  0,  0,  96,  0,  0,  0,  0}, //3
 {0,  0,  0,  96,  0,  0,  0,  0,  0,  0,  0,  0}, //4
 {0,  0,  0,  0,  0,  0,  96,  0,  0,  0,  0,  0}, //5
 {0,  0,  0,  0,  0,  0,  0,  0,  0,  96,  0,  0}, //6
 {0,  0,  0,  0,  0,  0,  0,  0,  96,  0,  0,  0}, //7
 {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  64,  0}, //8
 {0,  0,  0,  0,  0,  0,  0,  0,  96,  0,  0,  0}, //9
 {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  64}, //10
 {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0}}; //11
```

Figure B.1. MWD application for PFMAP algorithm.

In PFMAP there are various inputs to the algorithm. The first one of these is the application to be mapped on a NoC architecture. In Figure B.1, MWD application is given. In order to use an application as an input to the PFMAP algorithm, Task Traffic Graph (TTG)(see Figure B.1b) is extracted from the application block diagram (see Figure B.1a). In a TTG, nodes represent the block components of the application and edges represent the communication request between these nodes. Numbers on the edges define the communication volume between task nodes in 10KBytes/second for the given application. By using TTG of the input application, its 2-D array representation (see Figure B.1c) is used as an input to the PFMAP algorithm. In Figure B.1c, non-zero numbers represent coefficient of communication volumes between tasks. For example, according to the Figure B.1c, there exists a communication flow with the average value of 640 KBytes/second from *Task Node 0* to *Task Node 1*, similarly 1280 KBytes/second from *Task Node 0* to *Task Node 4*. These values are obtained from the first row of 2-D array given in Figure B.1. That is, source task nodes are ordered in rows, destination nodes are ordered in columns.

The second input for PFMAP is the target NoC architecture. A sample NoC architecture and its representation as a text file can be found in Figure B.2. In Figure B.2b, the integer value (this is 24 for our case) given in the first row represents the number of directed links on the target NoC architecture. The rest of the lines show the physical connections and their directions for core pairs. For example, the fifth line

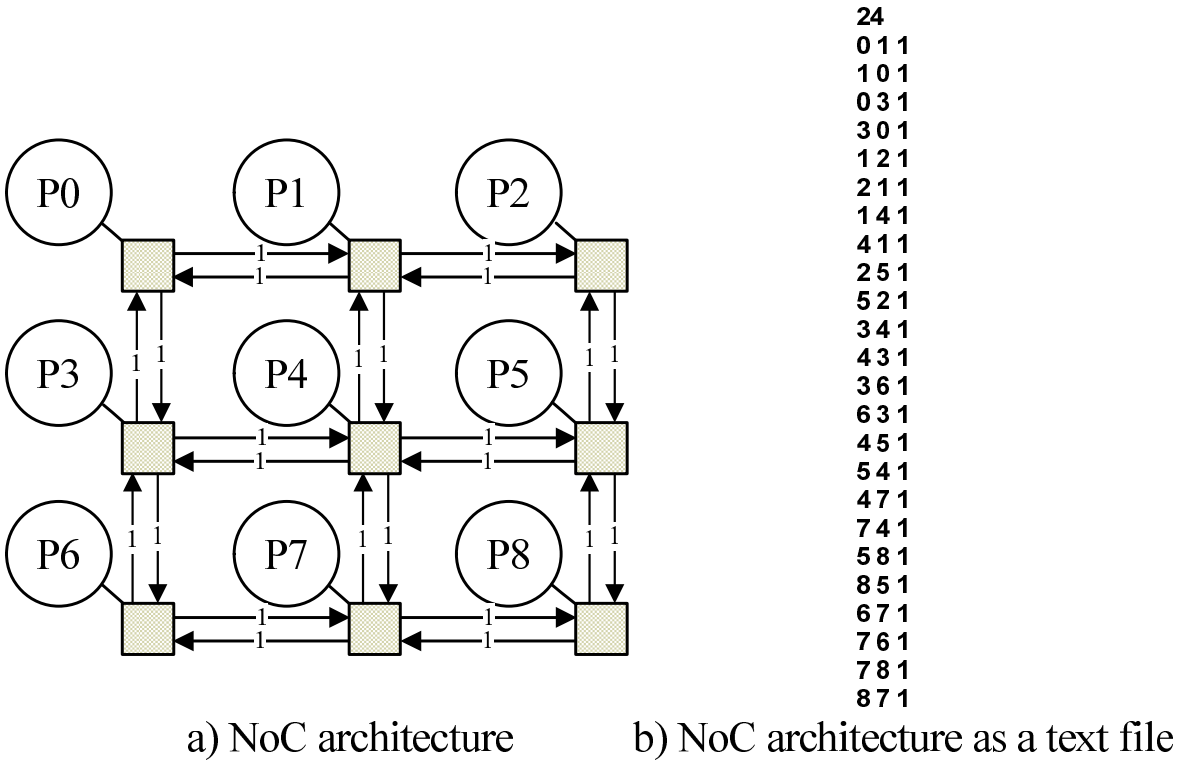


Figure B.2. A 2-D mesh 3x3 NoC architecture and its representation as a text file for PFMAP algorithm.

in Figure B.2b represents a direct connection from *Core 3* to *Core 0* with a bandwidth value of *1*. For regular NoC architectures, bandwidth value is always one (i.e. defining full bandwidth). For custom and irregular architectures, this value can be different from one.

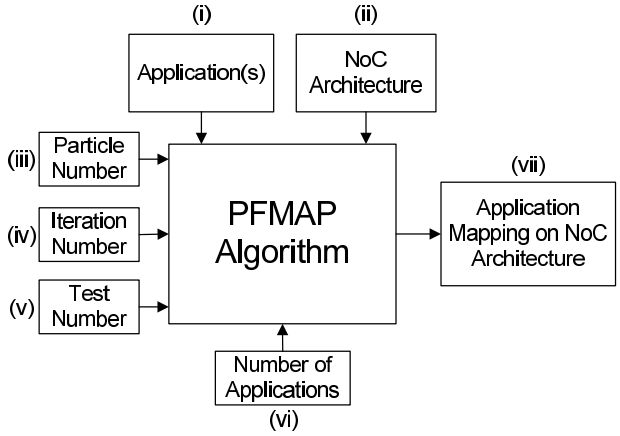


Figure B.3. Inputs and outputs of the PFMAP algorithm.

Inputs and output of the PFMAP algorithm is illustrated in Figure B.3. There are six inputs and one output for PFMAP algorithm. In addition to aforementioned

inputs there are also some numerical inputs for PFMAP. All of PFMAP inputs and output can be summarized as follows:

(i) **Application(s):** This input defines the application. In PFMAP, input task graph is represented as a 2-D double or integer array, where numbers represent coefficient of communication volumes between tasks. An example of such a 2-D array is given for MWD application in Figure B.1c. This 2-D array is defined in the header file of PFMAP algorithm. If there exists multiple applications, this input is a 3-D array, where each 2-D array element represents a different application.

(ii) **NoC Architecture:** This input defines the target NoC architecture. This input implemented as an external text file representing the communication links between cores. A text file representing, 2-D regular mesh NoC architecture with the size of 3x3 is given in Figure B.2b.

(iii) **Particle Number:** This number is an integer and defines the number of particles used for PFMAP.

(iv) **Iteration Number:** This number is an integer and defines the number of re-sampling iterations.

(v) **Test Number:** As PFMAP is a randomized algorithm, user can run the program multiple times and takes the average of results. Hence, we used such a parameter for PFMAP. This is also an integer number.

(vi) **Number of Applications:** As PFMAP can handle multiple input task graphs. Hence, there is an integer input, which defines the number of input tasks to the PFMAP algorithm. This input parameter is mostly used for scalability analysis of PFMAP.

(vii) **Application Mapping:** This output shows the final mapping result for the given application(s) on the target NoC architecture.

Input command format for PFMAP is given as follows:

PFMAP.exe APP IT PN TC NT NoC_Architecture_File_Name.txt

Here, the input parameters are given as in the following:

- *APP:* Input application name(s).
- *IT:* Number of iterations.
- *PN:* Number of particles.
- *TC:* Number of tests.
- *NAPP:* Number of applications.
- *NoC_Architecture_File_Name.txt:* NoC architecture representation as a text file (see Figure B.2b).

A sample input command for PFMAP is given as follows:

PFMAP.exe MWD 100 10000 100 1 dist_NoC_2D_Mesh_Reg_ROW_3_COL_4.txt

The corresponding output is kept in a text file. A sample output text file is given in Figure B.4. Here, the mapping found in each test is represented as a configuration. For example, in Figure B.4, first test result shows the configuration as follows:

- Task *2* is mapped onto the physical core *0*
- Task *3* is mapped onto the physical core *1*
- Task *11* is mapped onto the physical core *2*
- Task *10* is mapped onto the physical core *3*
- Task *1* is mapped onto the physical core *4*
- Task *4* is mapped onto the physical core *5*
- Task *7* is mapped onto the physical core *6*
- Task *8* is mapped onto the physical core *7*
- Task *5* is mapped onto the physical core *8*
- Task *0* is mapped onto the physical core *9*
- Task *6* is mapped onto the physical core *10*
- Task *9* is mapped onto the physical core *11*

```
exec. time GetCounter 32.429681 ms
Best configuration cost for 1. test: 1280
best configuration for 1. test is:

 2  3 11 10
 1  4  7  8
 5  0  6  9

exec. time GetCounter 28.079446 ms
Best configuration cost for 2. test: 1280
best configuration for 2. test is:

 1  2 11 10
 5  6  9  8
 0  4  3  7

...

exec. time GetCounter 27.814380 ms
Best configuration cost for 98. test: 1216
best configuration for 98. test is:

 5  6  9 11
 1  0  8 10
 2  4  7  3

exec. time GetCounter 28.109707 ms
Best configuration cost for 99. test: 1280
best configuration for 99. test is:

 2  1  5  6
 3  0  4  9
11 10  7  8

exec. time GetCounter 28.008200 ms
Best configuration cost for 100. test: 1216
best configuration for 100. test is:

11  9  6  5
10  8  0  1
 3  7  4  2

costLowerbound: 1120
The average exec time in 100 TESTs is: 28.696159
The best configuration cost in 100 TESTs is: 1216.000000
The average configuration cost in 100 TESTs is: 1264.960000
The worst configuration cost in 100 TESTs is: 1344
```

Figure B.4. PFMAP output text file for MWD application.

APPENDIX C: USER MANUAL FOR PFROUT

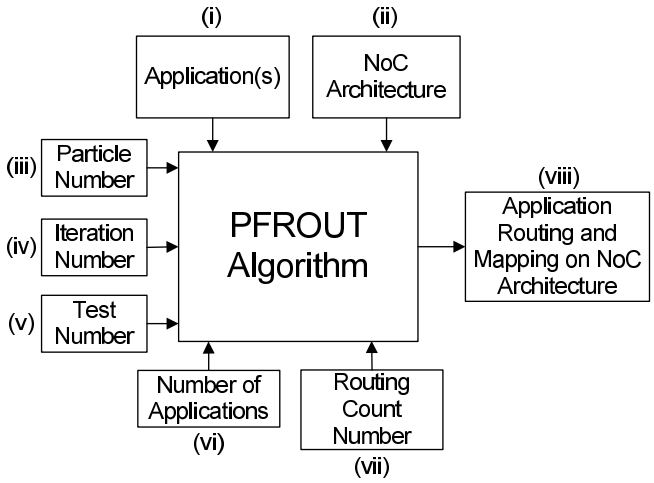


Figure C.1. Inputs and outputs of the PFROUT algorithm.

In PFROUT there are various inputs to the algorithm. Many of them are identical to the inputs given for PFMAP algorithm (see Appendix B). These are given as follows:

(i) **Application(s):** This input is the same input given in Figure B.1c for PFMAP.

(ii) **NoC Architecture:** This input is the same input given in Figure B.2b for PFMAP. However, there is also a parameter, which defines the Corridor Width (CW) of the target NoC architecture. This input parameter is defined in the header file of PFROUT algorithm. In PFROUT algorithm, we set CW parameter as either one or two.

(iii) **Particle Number:** This number is an integer and it defines the number of particles used for PFROUT.

(iv) **Iteration Number:** This number is an integer and it defines the number of re-sampling iterations.

(v) **Test Number:** Likewise PFMAP, PFROUT is also a randomized algorithm, user can run the program multiple times and takes the average of results. Hence, we also used such a parameter for PFROUT. This is also an integer number.

(vi) **Number of Applications:** Likewise PFMAP, PFROUT can handle multiple input task graphs. Hence, there is an integer input, which defines the number of input tasks to the PFROUT algorithm. This input parameter is mostly

used for scalability analysis of PFROUT (i.e. for running multiple TGFF3x3 to TGFF10x10 applications).

(vii) **Routing Count Number:** In path creation from a source node to a destination node, there are switches and routers. In PFROUT, by considering generated wavefront and the capacity of next router/switch, the direction is determined. At this step, if there exist multiple directions with the same cost, we always control the availability of the directions. Here, $RoutingCountNumber$ is a non-negative integer and it defines the number of routing iterations for a given mapping. If $RoutingCountNumber$ is set to zero then the control of the availability of the directions is always in the order of North-East-South-West. If $RoutingCountNumber$ is not zero, choosing of next router/switch in path creation is controlled each time in random order (e.g. South-East-West-North). If $RoutingCountNumber$ is set to N (e.g. N=10), then *findRouting* subroutine runs N times with random orders. In each run of *findRouting*, PFROUT tends to find a different routing for the same mapping. At the end, the best one of these routings is accepted as a solution.

(viii) **Application Routing and Mapping:** This output shows the final routing and mapping result for the given application(s) on the target NoC architecture. In addition to the mapping property of PFMAP, PFROUT also gives the routing information between communicating nodes on the given architecture.

Input command format for PFROUT is given as follows:

PFROUT.exe APP IT PN RC TC NT NoC_Architecture_File_Name.txt

Here, the input parameters are given as in the following:

- *APP:* Input application name(s).
- *IT:* Number of iterations.
- *PN:* Number of particles.
- *RC:* Routing count.
- *TC:* Number of tests.

- *NAPP:* Number of applications.
- *NoC_Architecture_File_Name.txt:* NoC architecture representation as a text file (see Figure B.2b).

A sample input command for PFROUT is given as follows:

PFROUT.exe TGFF3x3 10 100 0 10 1 dist_NoC_2D_Mesh_Reg_ROW_3_COL_3.txt

```
best configuration cost: 29177
best configuration is:
 1  0  2
 8  5  3
 7  6  4

1. path routing:
    0     0    0    0    0
    0     0    0    0    0
    0     0    0    0    0
    0     0    0    0    0
    0     0    4006 4006 4006


2. path routing:
    0     0    0    0    0
    0     0    0    0    0
    0     0    5006 0    0
    0     0    5006 0    0
    0     0    5006 0    0

…

14. path routing:
    0     0    0    2007 2007
    2007  2007 2007 2007 0
    2007  0    0    0    0
    2007  0    0    0    0
    2007  0    0    0    0
```

Figure C.2. PFROUT output text file for a synthetic TGFF3x3 application for CW=1.

The corresponding output is kept in a text file. A sample output text file is given in Figure C.2. Here, the mapping found for given applications is represented as a configuration. For example, in Figure C.2, best configuration is given as follows:

- Task *1* is mapped onto the physical core *0*
- Task *0* is mapped onto the physical core *1*
- Task *2* is mapped onto the physical core *2*
- Task *8* is mapped onto the physical core *3*
- Task *5* is mapped onto the physical core *4*
- Task *3* is mapped onto the physical core *5*
- Task *7* is mapped onto the physical core *6*
- Task *6* is mapped onto the physical core *7*
- Task *4* is mapped onto the physical core *8*

After giving mapping, PFROUT gives the routing between core pairs. Here, to identify paths, they are marked by using core numbers as follows:

$$PathMarkValue = (SourceCoreID \times 1000) + DestinationCoreID \tag{C.1}$$

For example, the 14^{th} path given in Figure C.2 defines the connection from *source core 2* to *destination core 7*.

REFERENCES

1. Modarressi, M., A. Tavakkol, and H. Sarbazi-Azad, "Application-Aware Topology Reconfiguration for On-Chip Networks", *IEEE Transactions on VLSI Systems*, Vol. 19, No. 11, pp. 2010–2022, 2011.

2. Dick, R. P., D. L. Rhodes, and W. Wolf, "TGFF: Task Graphs for Free", `http://ziyang.eecs.umich.edu/dickrp/tgff`, [Accessed: November 2013].

3. Lee, H. G., N. Chang, U. Y. Ogras, and R. Marculescu, "On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches", *ACM Transactions Design Automation Electronics Systems*, Vol. 12, pp. 23:1–23:20, 2008.

4. Tol, E. B. V. D. and E. G. T. Jaspers, "Mapping of MPEG-4 Decoding on a Flexible Architecture Platform", *Media Processors*, pp. 1–13, 2002.

5. Jang, W. and D. Z. Pan, "A3MAP: Architecture-Aware Analytic Mapping for Networks-on-Chip", *15th Asia and South Pacific Design Automation Conference*, pp. 523–528, 2010.

6. Jang, W. and D. Z. Pan, "A3MAP: Architecture-Aware Analytic Mapping for Networks-on-Chip", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 17, No. 3, p. 26, 2012.

7. Bayar, S. and A. Yurdakul, "PFMAP: Exploitation of Particle Filters for Network-on-Chip Mapping", *IEEE Transactions on VLSI Systems*, Vol. PP, No. NA, p. NA, 2014.

8. Bayar, S. and A. Yurdakul, "Dynamic Partial Self-Reconfiguration on Spartan-III FPGAs via a Parallel Configuration Access Port (PCAP)", *European Network of Excellence on High Performance and Embedded Architecture and Compilation*,

2008.

9. Bayar, S. and A. Yurdakul, "Self-Reconfiguration on Spartan-III FPGAs with Compressed Partial Bitstreams via a Parallel Configuration Access Port (cPCAP) Core", *Proceedings of the Ph. D. Research in Microelectronics and Electronics*, pp. 137–140, 2008.

10. "Support Document: XILINX", `http://www.xilinx.com/support/documentation`, [Accessed September 2010].

11. Carver, J., N. Pittman, and A. Forin, "Relocation of FPGA Partial Configuration Bit-Streams for Soft-Core Microprocessors", *Proceedings of the Workshop on Soft Processor Systems*, 2008.

12. Hilton, C. and B. Nelson, "PNoC: A Flexible Circuit-Switched NoC for FPGA-Based Systems", *Computers and Digital Techniques, IEE Proceedings* -, Vol. 153, No. 3, pp. 181 – 188, 2006.

13. Bobda, C., A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "DyNoC: A Dynamic Infrastructure for Communication In Dynamically Reconfugurable Devices", *International Conference on Field Programmable Logic and Applications*, pp. 153 – 158, 2005.

14. Stensgaard, M. and J. Sparso, "ReNoC: A Network-on-Chip Architecture with Reconfigurable Topology", *Second ACM/IEEE International Symposium on Networks-on-Chip*, pp. 55 –64, 2008.

15. Hollis, S. and C. Jackson, "Skip the Analysis: Self-Optimising Networks-on-Chip (Invited Paper)", *International Symposium on Electronic System Design*, pp. 14 –19, 2010.

16. Ma, J., C. Wang, Y. Wen, T. Chen, W. Hu, and J. Chen, "Dynamic Reconfigurable Networks in NoC for I/O Supported Parallel Applications", *International*

Conference on Computer and Information Technology, Vol. 0, pp. 2768–2775, 2010.

17. Modarressi, M., A. Tavakkol, and H. Sarbazi-Azad, "Virtual Point-to-Point Connections for NoCs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 29, No. 6, pp. 855 –868, 2010.

18. Vancayseele, R., B. Farisi, W. Heirman, K. Bruneel, and D. Stroobandt, "RecoNoC: A Reconfigurable Network-on-Chip", *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, pp. 1 –2, 2011.

19. Bayar, S. and A. Yurdakul, "A Dynamically Reconfigurable Communication Architecture for Multicore Embedded Systems", *Journal of Systems Architecture*, Vol. 58, pp. 140 – 159, 2012.

20. Kao, X. C., "Benefits Of Partial Reconfiguration", *Xcell Journal*, pp. 65–67, 2005.

21. Bayar, S., A. Yurdakul, and M. Tukel, "A Self-Reconfigurable Platform for General Purpose Image Processing Systems on Low-Cost Spartan-6 FPGAs", *Proceedings of the 6th Int Reconfigurable Communication-centric Systems-on-Chip Workshop*, pp. 1–9, 2011.

22. Bayar, S. and A. Yurdakul, "PFROUT: Particle Filtering Based Simultaneous Mapping and Routing Algorithm for Network-on-Chips", to be submitted.

23. Paulsson, K., M. Hübner, S. Bayar, and J. Becker, "Exploitation of Run-Time Partial Reconfiguration for Dynamic Power Management in Xilinx Spartan III-based Systems", *Reconfigurable Communication-centric Systems-on-Chip*, 2007.

24. Liu, S., R. N. Pittman, and A. Forin, "Energy Reduction with Run-Time Partial Reconfiguration", MSR-TR-2009- 2017, Technical Report of Microsoft Research, 2009.

25. Osborne, W., W. Luk, J. Coutinho, and O. Mencer, "Energy Reduction by Sys-

tematic Run-Time Reconfigurable Hardware Deactivation", *Transactions on European Network of Excellence on High Performance and Embedded Architecture and Compilation*, Vol. 4, No. 4, 2009.

26. Lorenz, M., L. Mengibar, M. Garcia-Valderas, and L. Entrena, "Power Consumption Reduction Through Dynamic Reconfiguration", *International Conference on Field Programmable Logic and Applications*, pp. 751–760, 2004.

27. Paulsson, K., M. Hübner, G. Auer, M. Dreschmann, L. Chen, and J. Becker, "Implementation of a Virtual Internal Configuration Access Port (JCAP) for enabling Partial Self-Reconfiguration on Xilinx Spartan-III FPGAs", *International Conference on Field Programmable Logic and Applications*, 2007.

28. Gonzalez, I., E. Aguayo, and S. Lopez-Buedo, "Self-Reconfigurable Embedded Systems on Low-Cost FPGAs", *IEEE Micro*, pp. 49 – 57, 2007.

29. Paulsson, K., U. Viereck, M. Hübner, and J. Becker, "Exploitation of the External JTAG Interface for Internally Controlled Configuration Readback and Self-Reconfiguration of Spartan 3 FPGAs.", *IEEE Computer Society Annual Symposium on VLSI*, pp. 304–309, 2008.

30. Claus, C., B. Zhang, W. Stechele, L. Braun, M. Hübner, and J. Becker, "A Multi-Platform Controller allowing for Maximum Dynamic Partial Reconfiguration Throughput.", *International Conference on Field Programmable Logic and Applications*, pp. 535–538, 2008.

31. Liu, M., W. Kuehn, Z. Lu, and A. Jantsch, "Run-Time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration", *Proceedings of the 19th International Conference on Field Programmable Logic and Applications*, pp. 498–502, 2009.

32. Koch, D., C. Beckhoff, and J. Torrison, "Advanced Partial Run-time Reconfiguration on Spartan-6 FPGAs", *Proceedings of International Conference on Field-*

Programmable Technology, 2010.

33. Bhandari, S. U., S. S. S. Pujari, and R. Mahajan, "Internal Dynamic Partial Reconfiguration for Real Time Signal Processing on FPGA", *Indian Journal of Science and Technology*, Vol. 3, No. 4, pp. 365–368, 2010.

34. Hübner, M., Diana, Göhringer, J. Noguera, and J. Becker, "Fast Dynamic and Partial Reconfiguration Data Path with Low Hardware Overhead on Xilinx FPGAs", *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, 2010.

35. Duhem, F., F. Muller, and P. Lorenzini, "FaRM: Fast Reconfiguration Manager for Reducing Reconfiguration Time Overhead on FPGA", *Proceedings of the 7th International Symposium on Applied Reconfigurable Computing*, pp. 23–25, 2011.

36. Silva, M. L. and J. C. Ferreira, "Support for Partial Run-Time Reconfiguration of Platform FPGAs", *Journal of Systems Architecture*, Vol. 52, No. 12, pp. 709–726, 2006.

37. Edwards, M. and P. Green, "Run-time Support for Dynamically Reconfigurable Computing Systems", *Journal of Systems Architecture*, Vol. 49, No. 4-6, pp. 267–281, 2003.

38. Ahmad, A., B. Krill, A. Amira, and H. Rabah, "Efficient Architectures for 3D HWT using Dynamic Partial Reconfiguration", *Journal of Systems Architecture.*, Vol. 56, No. 8, pp. 305–316, 2010.

39. Raghuraman, K. P., H. Wang, and S. Tragoudas, "A Novel Approach to Minimizing Reconfiguration Cost for LUT-Based FPGAs", *Proceedings of the 18th International Conference on VLSI Design: Power Aware Design of VLSI Systems*, pp. 673–676, 2005.

40. Rullmann, M. and R. Merker, "A Reconfiguration Aware Circuit Mapper for

FPGAs", *Proceedings of the 21st IEEE International Symposium on Parallel and Distributed Processing*, 2007.

41. Chen, W., Y. Wang, X. Wang, and C. Peng, "A New Placement Approach to Minimizing FPGA Reconfiguration Data", *International Conference on Embedded Software and Systems Symposia*, pp. 169–174, 2008.

42. Mehdipour, F., M. S. Zamani, H. R. Ahmadifar, M. Sedighi, and K. Murakami, "Reducing Reconfiguration Time of Reconfigurable Computing Systems in Integrated Temporal Partitioning and Physical Design Framework", *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, pp. 308–308, 2006.

43. Stepien, P. and M. Vasilko, "On Feasibility of FPGA Bitstream Compression During Placement and Routing", *International Conference on Field Programmable Logic and Applications*, pp. 1–4, 2006.

44. Sellers, B., J. Heiner, M. J. Wirthlin, and J. Kalb, "Bitstream Compression Through Frame Removal and Partial reconfiguration.", *International Conference on Field Programmable Logic and Applications*, pp. 476–480, 2009.

45. Li, Z. and S. Hauck, "Configuration Compression for Virtex FPGAs", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 147–159, 2001.

46. Gu, H. and S. Chen, "Partial Reconfiguration Bitstream Compression for Virtex FPGAs", *Image and Signal Processing, Congress on*, Vol. 5, pp. 183–185, 2008.

47. Koch, D. and J. Teich, "Platform Independent Methodology for Partial Reconfiguration", *Proceedings of the 1st Conference on Computing Frontiers*, 2004.

48. Ştefan, R. and S. D. Coţofană, "Bitstream Compression Techniques For Virtex 4 FPGAS", *International Conference on Field Programmable Logic and Applications*, pp. 323–328, 2008.

49. Koch, D., C. Beckhoff, and J. Teich, "Bitstream Decompression for High Speed FPGA Configuration from Slow Memories", *Proceedings of International Conference on Field-Programmable Technology*, pp. 323–328, 2007.

50. Dandalis, A. and V. K. Prasanna, "Configuration Compression for FPGA-Based Embedded Systems", *IEEE Transactions on VLSI Systems*, Vol. 13, No. 12, pp. 1394 – 1398, 2005.

51. Xilinx, "The Low-Cost Spartan-6 FPGA Family Delivers an Optimal Balance of Low Risk, Low Cost, Low Power, and High Performance", Company Release, 2009.

52. Pasricha, S. and N. Dutt, *On-Chip Communication Architectures: System on Chip Interconnect*, Morgan Kaufmann Publishers Inc., 2008.

53. Koziris, N., M. Romesis, P. Tsanakas, and G. Papakonstantinou, "An Efficient Algorithm for the Physical Mapping of Clustered Task Graphs onto Multiprocessor Architectures", *Proceedings of the 8th Euromicro Workshop Parallel and Distributed Processing*, pp. 406–413, 2000.

54. Murali, S. and G. De Micheli, "Bandwidth-Constrained Mapping of Cores onto NoC Architectures", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Vol. 2, pp. 896–901, 2004.

55. Hu, J. and R. Marculescu, "Energy-Aware Mapping for Tile-Based NoC Architectures Under Performance Constraints", *8th Asia and South Pacific Design Automation Conference*, pp. 233–239, 2003.

56. Murali, S. and G. De Micheli, "SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs", *Proceeding of the 41st Design Automation Conference*, pp. 914–919, 2004.

57. Moein-darbari, F., A. Khademzade, and G. Gharooni-fard, "CGMAP: A New

Approach to Network-on-Chip Mapping Problem", *Institute of Electronics, Information and Communication Engineers Electronics Express*, Vol. 6, No. 1, pp. 27–34, 2009.

58. Janidarmian, M., A. Khademzadeh, and M. Tavanpour, "Onyx: A New Heuristic Bandwidth-Constrained Mapping of Cores onto Tile-Based Network on Chip", *Institute of Electronics, Information and Communication Engineers Electronics Express*, Vol. 6, No. 1, pp. 1–7, 2009.

59. Saeidi, S., A. Khademzadeh, and F. Vardi, "Crinkle: A Heuristic Mapping Algorithm for Network-on-Chip", *Institute of Electronics, Information and Communication Engineers Electronics Express*, Vol. 6, No. 24, pp. 1737–1744, 2009.

60. Zhong, L., J. Sheng, M. Jing, Z. Yu, X. Zeng, and D. Zhou, "An Optimized Mapping Algorithm Based on Simulated Annealing for Regular NoC Architecture", *Proceedings of the IEEE 9th Internation Application Specific Integrated Circuit Conference*, pp. 389–392, 2011.

61. Sahu, P. K. and S. Chattopadhyay, "A Survey on Application Mapping Strategies for Network-on-Chip Design", *Journal of Systems Architecture*, Vol. 59, No. 1, pp. 60 – 76, 2013.

62. Tosun, S., O. Ozturk, and M. Ozen, "An ILP Formulation for Application Mapping onto Network-on-Chips", *International Conference on Application of Information and Communication Technologies*, pp. 1–5, 2009.

63. Sahu, P., N. Shah, K. Manna, and S. Chattopadhyay, "A New Application Mapping Algorithm for Mesh Based Network-on-Chip Design", *Annual IEEE India Conference*, pp. 1–4, 2010.

64. Sahu, P., P. Venkatesh, S. Gollapalli, and S. Chattopadhyay, "Application Mapping onto Mesh Structured Network-on-Chip Using Particle Swarm Optimization", *IEEE Computer Society Annual Symposium on VLSI*, pp. 335–336, 2011.

65. Dick, R. P., "Embedded System Synthesis Benchmarks Suites (E3S)", `http://ziyang.eecs.umich.edu/dickrp/e3s`, [Accessed: November 2013].

66. Ogras, U. and R. Marculescu, "Application-Specific Network-on-Chip Architecture Customization via Long-Range Link Insertion", *IEEE/ACM International Conference on Computer-Aided Design*, pp. 246 – 253, 2005.

67. Giefers, H. and M. Platzner, "A Triple Hybrid Interconnect for Many-Cores: Reconfigurable Mesh, NoC and Barrier", *International Conference on Field Programmable Logic and Applications*, pp. 223 –228, 2010.

68. Modarressi, M., M. Asadinia, and H. Sarbazi-Azad, "Using Task Migration to Improve Non-Contiguous Processor Allocation in NoC-Based CMPs", *Journal of Systems Architecture*, Vol. 59, No. 7, pp. 468 – 481, 2013.

69. Woo, S. C., M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", *The ACM Special Interest Group on Computer Architecture News*, Vol. 23, No. 2, pp. 24 – 36, 1995.

70. Teimouri, N., M. Modarressi, and H. Sarbazi-Azad, "Power and Performance Efficient Partial Circuits in Packet-Switched Networks-on-Chip", *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 509–513, 2013.

71. Wang, C. and N. Bagherzadeh, "Design and Evaluation of a High Throughput QoS-Aware and Congestion-Aware Router Architecture for Network-on-Chip", *Microprocessors and Microsystems*, Vol. 38, No. 4, pp. 304 – 315, 2014.

72. Swaminathan, K., S. G. Nambiar, Rajkumar, G. Lakshminarayanan, and S. Ko, "A Novel Hybrid Topology for Network on Chip", *IEEE 27th Canadian Conference on Electrical and Computer Engineering*, pp. 1–6, 2014.

73. Lee, J., C. Nicopoulos, H. Lee, and J. Kim, "Centaur: A Hybrid Network-on-Chip Architecture Utilizing Micro-Network Fusion", *Design Automation for Embedded Systems*, pp. 1–19, 2014.

74. Bruneel, K., F. Abouelella, and D. Stroobandt, "Automatically Mapping Applications to a Self-Reconfiguring Platform", *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 964–969, 2009.

75. Bjerregaard, T. and S. Mahadevan, "A Survey of Research and Practices of Network-on-chip", *ACM Computing Surveys*, Vol. 38, No. 1, 2006.

76. Gustafsson, F., F. Gunnarsson, N. Bergman, U. Forssell, J. Jansson, R. Karlsson, and P. Nordlund, "Particle Filters for Positioning, Navigation, and Tracking.", *IEEE Transactions on Signal Processing*, Vol. 50, No. 2, pp. 425–437, 2002.

77. Thrun, S., "Particle Filters in Robotics", *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence*, pp. 511–518, 2002.

78. Rekleitis, I., "A Particle Filter Tutorial for Mobile Robot Localization", Technical report, Centre for Intelligent Machines, McGill University, 2004.

79. Doucet, A., N. De Freitas, and N. Gordon, *Sequential Monte Carlo Methods in Practice*, Springer New York, 2001.

80. Gordon, N. J., D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation", *Radar and Signal Processing, IEE Proc. F*, Vol. 140, No. 2, 1993.

81. Douc, R., O. Cappé, and E. Moulines, "Comparison of Resampling Schemes for Particle Filtering", *4th International Symposium on Imgage and Signal Processing and Analysis*, 2005.

82. Kitagawa, G., "Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models", *Journal of Computational and Graphical Statistics*, Vol. 5,

No. 1, pp. 1–25, 1996.

83. Hol, J., T. Schon, and F. Gustafsson, "On Resampling Algorithms for Particle Filters", *Nonlinear Statistical Signal Processing Workshop*, pp. 79–82, 2006.

84. Dijkstra, E., "A Note on Two Problems in Connexion with Graphs", *Numerische Mathematik*, Vol. 1, No. 1, pp. 269–271, 1959.

85. Fisher, R. A. and F. Yates, *Statistical Tables for Biological, Agricultural and Medical Research*, Oliver and Boyd, 3rd edition, 1948.

86. Saito, M. and M. Matsumoto, "SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator", *Monte Carlo and Quasi-Monte Carlo Methods*, pp. 607–622, 2008.

87. Tian, X. and K. Benkrid, "Mersenne Twister Random Number Generation on FPGA, CPU and GPU", *NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 460–464, 2009.

88. Matsumoto, M. and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator", *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1, pp. 3–30, 1998.

89. OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.0", `http://www.openmp.org/mp-documents/spec30.pdf`, [Accessed September 2012].

90. "NIRGAM (v2.0)", `http://www.nirgam.ecs.soton.ac.uk`, [Accessed March 2012].

91. Hu, J. and R. Marculescu, "Energy- and Performance-Aware Mapping for Regular NoC Architectures", *Computer-Aided Design of Integrated Circuits and Systems*, Vol. 24, No. 4, pp. 551–562, 2005.

92. He, O., S. Dong, W. Jang, J. Bian, and D. Pan, "UNISM: Unified Scheduling and Mapping for General Networks on Chip.", *IEEE Transactions on VLSI Systems*, Vol. 20, No. 8, pp. 1496–1509, 2012.

93. Liu, C., L. Zhang, Y. Han, and X. Li, "Vertical Interconnects Squeezing in Symmetric 3D Mesh Network-on-Chip", *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pp. 357–362, 2011.

94. Wang, X., P. Liu, M. Yang, M. Palesi, Y. Jiang, and M. Huang, "Energy Efficient Run-Time Incremental Mapping for 3-D Networks-on-Chip", *Journal of Computer Science and Technology*, Vol. 28, No. 1, pp. 54–71, 2013.

95. Schonwald, T., A. Viehl, O. Bringmann, and W. Rosenstiel, "Distance-Constrained Force-Directed Process Mapping for MPSoC Architectures", *5th Euromicro Conference on Digital Systems Design*, 2012.

96. Young Hur, J., S. Wong, and S. Vassiliadis, "Partially Reconfigurable Point-to-Point FPGA Interconnects", *International Journal of Electronics*, Vol. 95, No. 7, pp. 725–742, 2008.

97. "NoCem: Network on Chip Emulator", `http://www.opencores.org/nocem`, [Accessed June 2011].

98. Schelle, G. and D. Grunwald, "Onchip Interconnect Exploration for Multicore Processors Utilizing FPGAs", *In 2nd Workshop on Architecture Research using FPGA Platforms*, pp. 1–4, 2006.

99. Srinivasan, M., M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli, "A Methodology for Mapping Multiple Use-Cases onto Networks on Chips", *Proceedings of the Design, Automation and Test in Europe*, Vol. 1, pp. 1–6, 2006.

100. Huebner, M., T. Becker, and J. Becker, "Real-time LUT-based Network Topologies for Dynamic and Partial FPGA Self-reconfiguration", *Proceedings of the 17th*

Symposium on Integrated Circuits and System Design, pp. 28–32, 2004.

101. Möller, L., R. Soares, E. Carvalho, I. Grehs, N. Calazans, and F. Moraes, "Infrastructure for Dynamic Reconfigurable Systems: Choices and Trade-offs", *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design*, pp. 44–49, 2006.

102. Khan, J., S. Niar, M. A. R. Saghir, Y. Elhillali, and A. Rivenq-Menhaj, "Trade-Off Exploration for Target Tracking Application in a Customized Multiprocessor Architecture.", *EURASIP Journal on Embedded Systems*, Vol. 2009, 2009.

103. Asanovic, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley", Technical report, EECS Department, University of California, 2006.

104. Demiroz, B., H. Topcuoglu, M. Kandemir, and O. Tosun, "Parallelizing Barnes-Hut Method on the Cell BE Architecture", *3rd Workshop on Programmability Issues for Heterogeneous Multicores*, 2010.

105. Kistler, M., M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed", *IEEE Micro*, Vol. 26, No. 3, pp. 10–23, 2006.

106. Welsh, D. J. A. and M. B. Powell, "An Upper Bound for the Chromatic Number of a Graph and its Application to Timetabling Problems", *The Computer Journal*, Vol. 10, No. 1, pp. 85–86, 1967.