System-Level Synthesis of Dataplane Subsystems for MPSoCs

by

Glenn Leary

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved May 2013 by the
Graduate Supervisory Committee:

ARIZONA STATE UNIVERSITY

August 2013

ABSTRACT

In recent years we have witnessed a shift towards multi-processor system-on-chips (MPSoCs) to address the demands of embedded devices (such as cell phones, GPS devices, luxury car features, etc.). Highly optimized MPSoCs are well-suited to tackle the complex application demands desired by the end user customer. These MPSoCs incorporate a constellation of heterogeneous processing elements (PEs) (general purpose PEs and application-specific integrated circuits (ASICS)). A typical MPSoC will be composed of a application processor, such as an ARM Coretex-A9 with cache coherent memory hierarchy, and several application sub-systems. Each of these sub-systems are composed of highly optimized instruction processors, graphics/DSP processors, and custom hardware accelerators. Typically, these sub-systems utilize scratchpad memories (SPM) rather than support cache coherency. The overall architecture is an integration of the various sub-systems through a high bandwidth system-level interconnect (such as a Network-on-Chip (NoC)). The shift to MPSoCs has been fueled by three major factors: demand for high performance, the use of component libraries, and short design turn around time. As customers continue to desire more and more complex applications on their embedded devices the performance demand for these devices continues to increase. Designers have turned to using MPSoCs to address this demand. By using pre-made IP libraries designers can quickly piece together a MPSoC that will meet the application demands of the end user with minimal time spent designing new hardware. Additionally, the use of MPSoCs allows designers to generate new devices very quickly and thus reducing the time to market. In this work, a complete MPSoC synthesis design flow is presented.

We first present a technique [23] to address the synthesis of the interconnect architecture (particularly Network-on-Chip (NoC)). We then address the synthesis of the memory architecture of a MPSoC sub-system [24]. Lastly, we present a co-synthesis technique to generate the functional and memory architectures simultaneously. The validity and quality of each synthesis technique is demonstrated through extensive experimentation.

DEDICATION

To my wife, Amanda, for enduring this long road with me. I Love You.

To my sweet little angel, Lauren, you have changed my life forever.

And to my Zoey. You were truly a Man's best friend. I will forever miss you.

Zoey
8.1.2011 - 1.18.2013

ACKNOWLEDGEMENTS

First I would like to give my thanks to my graduate advisor Dr. Karam Chatha for directing me throughout my Ph.D. study. It was his guidance that led me into the field of embedded systems. His knowledge and expertise has given me the guidance to overcome the obstacles that I encountered during my study. His help and insight has contributed to almost all of my research and the papers that I have published. I am honored to have had the opportunity to work with Dr. Chatha during the 6+ years (that's a long time) of my Ph.D. study. His wisdom will continue to guide me and help me throughout my professional career.

Secondly, I would like to thank my committee members for agreeing to join my committee. I sincerely appreciate their time and guidance. I would like to thank Dr. Vrudhula for his active participation in discussions and his dedication to the Embedded Systems Consortium. I would like to thank Dr. Shrivastava for his active role and contributions to the computing systems research lab. I would like to thank Dr. Beraha at Qualcomm Inc. for his professional insight into embedded devices. I consider myself lucky to have such a kind and knowledgable committee.

I would like to thank all of the lab members that I have spent time with, discussed research problems with, conducted projects with me, collaborated on papers with me, and who have made the process more enjoyable. I would like to thank Michael Baker for discussing research problems with me, working on projects with me, traveling the globe with me, and making life more enjoyable in and out of the lab. I would like to thank Sushu Zhang for her support at the beginning when I was freshly starting my journey. I would like to thank Weijia Che for his help with the problems I had with Cell BE and StreamIT. I would like to thank Amrit Panda for his knowledge and the discussions we have had. I would to thank Jyothi Swaroop for his collaboration on SystemC. There have been many other lab members that have been of great help: Nikhil Ghadge, HaeSeung Lee, Krishna Mehta, and many others. They have all helped me in one way or another and I would like to thank them for their time and commitment.

I would like to thank the support I have received from outside of Arizona State University. In particular, the collaboration with the IUCRC has given me insight into the industry world and introduced me to many people. The collaboration with Qualcomm Inc. has been a great help in both leading me technically and supporting my study financially. The task of developing a Network-on-Chip synthesis technique for their IP has given me insight into a part of the industry I was blind to prior. The collaborations with the industry partners not only helped me be successful during my academic studies but also helped lay the foundation for a bountiful future after my graduation.

Last but not least, I would like to thank my family. I would like to thank my parents, brother, and sister for always being there for me when I needed it the most. I can not express in words what it has meant to have you all by my side. I would also like to thank my wife for sticking by me through it all. It has been a long and tenuous journey but you were always there for me. I love you and I am so happy to have you at my side as we begin the next chapter in our life together.

TABLE OF CONTENTS

Page

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER

1 INTRODUCTION 1

1.1 Contributions 5

2 THE SYNTHESIS OF THE NETWORK-ON-CHIP ARCHITECTURE 6

2.1 Motivation 7

2.2 Problem description 8

2.3 Related Work 10

2.4 NoC Synthesis Technique 11

2.4.1 Design flow 11

2.4.2 Basic approach to topology synthesis 12

2.4.3 Deadlock avoidance 14

2.4.4 Communication requirements 19

2.4.5 Port arity constraints 23

2.4.6 Multiple use cases 25

2.4.7 Algorithm Time Complexity Analysis 25

2.5 Experimental Results 26
2.5.1 Experimentation set-up 26
Benchmark designs 26
Existing approaches 27
Evaluation metrics 28
2.5.2 Comparisons with existing approaches 29
Comparison with ILP for $S1$ 29
Comparison with GT for $S2$ 31
Summary of comparisons with ILP and GT . . . 33
2.5.3 Impact of port arity 37
2.5.4 Impact of increasing bandwidth 38
2.5.5 Degree of transaction interference 41
2.5.6 Deadlock avoidance analysis 43
2.6 Summary 44
3 THE SYNTHESIS OF THE MEMORY ARCHITECTURE 47
3.0.1 Motivation 47
3.1 Problem Definition 52
3.2 Related Work 54
3.3 System-level memory synthesis 56

3.3.1 Data memory minimization 58
3.3.2 Code memory minimization 60
3.3.3 Time Complexity Analysis 62
3.4 Experimental Results 63
3.4.1 Comparison against Baseline Solution 63
3.4.2 Impact of Code Overlay Optimization 66
3.4.3 Comparison with Existing Approach 67
3.4.4 Impact of Area Constraint 69
3.5 Summary 70
4 THE SYNTHESIS OF THE FUNCTIONAL ARCHITECTURE . 71
4.1 Motivation 71
4.2 Problem Definition 75
4.3 Related Work 76
4.4 Functional Architecture Synthesis 80
4.4.1 Initial Solution 80
4.4.2 Data Memory Minimization 83
4.4.3 Code Memory Minimization 86
4.4.4 Hardware Accelerator Replication 89
4.4.5 Iterative Transition to HW-SW Design 90

CHAPTER Page

4.5 Experimental Results 93

4.5.1 High Level Synthesis of Hardware Accelerator Library . 94

4.5.2 DCT Comparison 95

4.5.3 FFT Comparison 98

4.5.4 Impact of Overlay 102

4.6 Extended Results 107

4.7 Summary 128

5 CONCLUSION AND FUTURE WORK 129

5.0.1 Virtual Platform 130

5.0.2 RTL Generation 131

REFERENCES 132

APPENDIX

A NETWORK-ON-CHIP SYNTHESIS 137

B MEMORY ARCHITECTURE SYNTHESIS 145

C FUNCTIONAL ARCHITECTURE SYNTHESIS 151

LIST OF TABLES

Table — Page

2.1 Categories of synthetic benchmarks 26

3.1 Benchmark Specifications 62

4.1 Minimum Buffer PASS Generation Sequence 81

4.2 Least Switching PASS Generation Sequence 82

4.3 Data Inteference Table 85

4.4 Code Inteference Table 87

4.5 Hardware Accelerators Before Replication 89

4.6 Hardware Accelerators After Replication 90

4.7 Benchmark Specifications 93

4.8 Benchmark Constraints 94

4.9 Single Software Core Comparison (FFT) 105

LIST OF FIGURES

Figure Page

1.1 Generic MPSoC Architecture 2

1.2 System-Level Synthesis Flow 2

2.1 Transaction level specification 9

2.2 Overall design flow 12

2.3 Shortest path graphs 13

2.4 NoC Topology Synthesis 14

2.5 Example of a Deadlock 15

2.6 Deadlock Example - SPGs 15

2.7 Deadlock Example - Global SPG 16

2.8 Deadlock Example - Global CDG 17

2.9 Deadlock Example - Updated SPGs 18

2.10 Pseudocode for deadlock avoidance 19

2.11 Bandwidth violation 20

2.12 Transaction interference 21

2.13 Pseudocode for trace conflict resolution 22

2.14 Router arity constraints 23

2.15 Pseudocode for router port arity resolution 24

2.16 Power/Router for ILP 29

2.17 Avg. packet latency 30

2.18 Avg. standard deviation 31

2.19 Power/Router for GT 32

2.20 Avg. packet latency 33

2.21 Avg. standard deviation 34

2.22 Power/router comparisons 35

Figure Page

2.23 Latency/std. dev. comparisons 36
2.24 Power/Routers versus Port Arity 37
2.25 Power Consumption with Increasing Bandwidth 38
2.26 Required Routers with Increasing Bandwidth 39
2.27 Reduction in latency and standard deviation 40
2.28 Normalized Power and Routers with Increasing Transaction Overlap 41
2.29 Average Packet Latency with Increasing Transaction Overlap . . . 42
2.30 Average Standard Deviation with Increasing Transaction Overlap 43
2.31 Application specification 44
2.32 ILP solution 45
2.33 Our solution 45
3.1 Generic MPSoC architecture 48
3.2 System-level design flow 49
3.3 Architecture of MPSoC sub-system 50
3.4 Top-level view of memory synthesis 55
3.5 Data memory minimization pseudo-code 58
3.6 Clique partitioning pseudo-code 59
3.7 Code memory minimization pseudo-code 61
3.8 Normalized area 64
3.9 Normalized throughput 65
3.10 Normalized Perf./Watt 66
3.11 Impact of code overlay 67
3.12 Existing approach 68
3.13 Area impact 69

Figure Page

4.1 Generic MPSoC architecture 72

4.2 System-level design flow 73

4.3 Architecture of MPSoC sub-system 74

4.4 Functional Architecture Synthesis Flowchart 79

4.5 Example SDF Specification 80

4.6 Data Memory Minimization Pseudo-code 84

4.7 Example Data Lifetimes 85

4.8 Data Overlay Schedule 86

4.9 Code Memory Minimization Pseudo-code 87

4.10 Example SDF Specification 87

4.11 Least Switching Schedule 88

4.12 Hardware Accelerator Replication Pseudo-code 88

4.13 Iterative Transition to HW-SW Design Pseudo-code 90

4.14 DCT Area vs. Perf. vs. Energy (sb) 96

4.15 DCT Area vs. Perf. vs. Energy (sb) 97

4.16 DCT Area vs. Perf. vs. Energy (ls) 98

4.17 DCT Area vs. Perf. vs. Energy (ls) 99

4.18 FFT Area vs. Perf. vs. Energy (sb) 100

4.19 FFT Area vs. Perf. vs. Energy (sb) 101

4.20 FFT Area vs. Perf. vs. Energy (ls) 102

4.21 FFT Area vs. Perf. vs. Energy (ls) 103

4.22 DCT Overlay Impact (sb) 104

4.23 DCT Overlay Impact (ls) 105

4.24 beamformer Area vs. Perf. vs. Energy (sb) 108

Figure Page

4.25 beamformer Area vs. Perf. vs. Energy (sb) 109
4.26 beamformer Area vs. Perf. vs. Energy (ls) 110
4.27 beamformer Area vs. Perf. vs. Energy (ls) 111
4.28 bitonic-sort Area vs. Perf. vs. Energy (sb) 112
4.29 bitonic-sort Area vs. Perf. vs. Energy (sb) 113
4.30 bitonic-sort Area vs. Perf. vs. Energy (ls) 114
4.31 bitonic-sort Area vs. Perf. vs. Energy (ls) 115
4.32 channelvocoder Area vs. Perf. vs. Energy (sb) 116
4.33 channelvocoder Area vs. Perf. vs. Energy (sb) 117
4.34 channelvocoder Area vs. Perf. vs. Energy (ls) 118
4.35 channelvocoder Area vs. Perf. vs. Energy (ls) 119
4.36 filterbank Area vs. Perf. vs. Energy (sb) 120
4.37 filterbank Area vs. Perf. vs. Energy (sb) 121
4.38 filterbank Area vs. Perf. vs. Energy (ls) 122
4.39 filterbank Area vs. Perf. vs. Energy (ls) 123
4.40 fm Area vs. Perf. vs. Energy (sb) 124
4.41 fm Area vs. Perf. vs. Energy (sb) 125
4.42 fm Area vs. Perf. vs. Energy (ls) 126
4.43 fm Area vs. Perf. vs. Energy (ls) 127

Chapter 1

INTRODUCTION

In recent years, multi-processor system-on-chips (MPSoCs) have emerged as the architecture of choice in embedded devices (such as cell phones, GPS devices, luxury car features, etc.) to address the complex applications desired by the end user customer. These MPSoCs incorporate a constellation of heterogeneous processing elements (PEs) (general purpose PEs and application-specific integrated circuits (ASICS)). As an example, Figure 1.1 depicts the top-level view of a generic architecture for a MPSoC. The application processor is a general purpose processor such as an ARM Cortex-A9 that supports a cache coherent memory hierarchy. The other application sub-systems are composed of highly optimized instruction processors, graphics/DSP processor, and custom hardware accelerators. Typically, the non-application sub-systems do not support a cache hierarchy and instead utilize scratchpad memories. The overall architecture is an integration of the various sub-systems through a high bandwidth system-level interconnect (such as an Network-on-Chip (NoC)).

The shift from single processor designs to MPSoCs has been fueled by three major factors: demand for high performance, the use of component libraries, and short design turn around time. As customers continue to desire more and more complex applications on their embedded devices the performance demand for these devices continues to increase. This increasing performance demand has become difficult for single core architectures to meet. Whereas, an MPSoC architecture is well suited to meet the performance

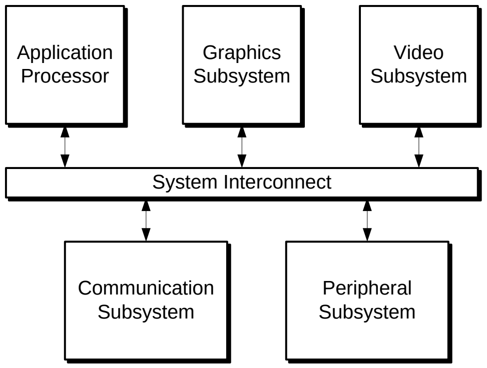


Figure 1.1: Generic MPSoC Architecture

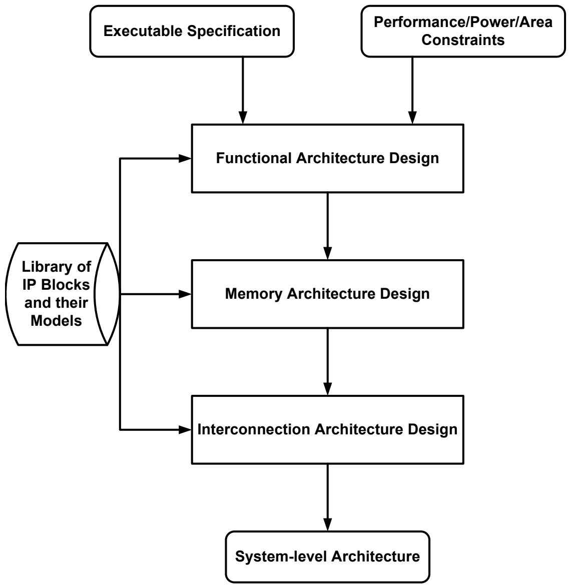

Executable Specification
Performance/Power/Area Constraints
Functional Architecture Design
Library of IP Blocks and their Models
Memory Architecture Design
Interconnection Architecture Design
System-level Architecture

Figure 1.2: System-Level Synthesis Flow

demand through the use of several high-performance sub-systems and concurrent on-chip communication.

The second factor leading to the shift to MPSoCs is the re-use of IP blocks by device designers. Designing new IP blocks takes a lot of time and money. With a single processor design this would be required far more often than with a MPSoC architecture. With a MPSoC approach, designers maintain a library of their available IP blocks. Designers then pick and choose from the IP blocks in order to build the sub-systems for new devices and to meet the performance demand. This therefore, reduces the frequency of the development of new IP.

The final factor leading to the shift towards MPSoCs is the short turn around time on embedded devices. This short turn around time is the direct result of the competitive environment of the industry. Each member in the industry is striving to maintain their edge over their competition. In order to achieve this, and meet consumer demands, companies are required to produce new higher performing devices at rapid rates. Due to designers using libraries of IP blocks and MPSoC architectures, they are able to build new devices with very quick turn around times.

Figure 1.2 illustrates the major stages in the system-level MPSoC design flow. The inputs to the design flow are an executable specification), a set of constraints (performance, power, and area), and a library of characterized IP blocks (performance, power, and area models). The design flow of the system architecture consists of three major stages: functional design, memory design, and interconnection design. During the functional architecture design stage the required processing elements (PEs) are selected and the functionality is mapped to these PEs. During the memory architecture design stage the number and configuration (size, number of ports, etc.) of the various memory elements are selected. And lastly, during the interconnection architecture design stage the underlying topology for the interconnect is specified.

Typically several sub-systems of a MPSoC (graphics, multimedia, communication, etc.) exhibit classic streaming behavior. Therefore, it is a natural choice to describe these sub-systems by utilizing stream programming formats. For the purpose of this work we assume that the functionality of the applications are described by a synchronous dataflow (SDF) specification [22]. A typical SDF specification will be represented through a graph consisting of nodes and arcs. Each node represents an actor and each arc represents the passing of data between actors through tokens. Each actor represents a section of the applications functionality (typically one or more filters within a streaming application). Each token in the graph represents a block of data. Each arc will be annotated with the number of tokens produced by the actor at the head of the arc and the number of tokens consumed by the actor at the tail of the arc.

As the complexity and performance demands of the applications on embedded devices continues to increase, it is becoming more difficult for designers to meet the imposed constraints within the short design turn around time with a manual design approach. In order to help with the design of the MPSoCs, designers have began to explore the automation of the process. The focus of this dissertation is on the automation of the complete system-level MPSoC design flow.

1.1 Contributions

The contributions of this dissertation are summarized as follows:

1. We present a novel technique for the synthesis of application specific Network-on-Chip (NoC) interconnect architectures (Chapter 2). The technique includes several design requirements: mixed communication traffic (cumulative/transactional), port arity constraints, deadlock avoidance, and multiple use-cases. The technique generates superior architectures than other existing techniques in terms of power consumption, area, and latency.

2. We present a novel technique for the synthesis of the memory architecture for a MPSoC sub-system for a given SDF specification (Chapter 3). The technique makes smart decisions to reduce both the code and data memory area with minimal performance degradation. The technique generates superior memory architectures than other existing techniques in terms of performance, area, and energy consumption.

3. We present a novel technique for the co-synthesis of the functional and memory architectures for a MPSoC sub-system for a given SDF specification (Chapter 4). The technique generates HW-SW designs that provide a desirable balance between the flexibility of software and the performance of hardware. The technique also simultaneously generates a memory architecture for the sub-system that makes smart decisions to reduce code and data memory area. The technique is shown to generate highly optimized and efficient designs in terms of performance, area, and energy consumption.

Chapter 2

THE SYNTHESIS OF THE NETWORK-ON-CHIP ARCHITECTURE

Application specific Network-on-Chip (NoC) architectures have emerged as a leading technology to address the communication woes of multi-processor System-on-Chip architectures. Synthesis approaches for custom NoC must address several requirements including cumulative bandwidth and transaction level (TL) communication requirements, multiple application use-cases, deadlock avoidance, and router port bandwidth and arity constraints. In this chapter we present a holistic algorithm for NoC synthesis which is able to address all these requirements together in an integrated manner. The approach is able to generate designs that consume minimum dynamic power consumption, and at most twice the number of routers (and leakage power) as an optimal solution. In terms of performance the technique is able to generate NoC designs with very low average communication latencies (verified by actual simulations) and equally low standard deviation (jitter) while utilizing simple best effort routers. We evaluated the effectiveness and quality of the proposed technique by comparisons with two existing approaches. Extensive experimental results are presented for synthetic/realistic multiple use case applications, cumulative/transaction traffic requirements, increasing application bandwidth requirements, and different port arity constraints.

In the next section we motivate the problem. Section 2.2 formally formulates the problm definition. We discuss related work in Section 2.3. We present the synthesis technique in Section 2.4. In Section 2.5 we present our experimental results. Lastly, we summarize the chapter in Section 2.6.

2.1 Motivation

Network-on-Chip (NoC) has been emerging as the solution of choice to address the challenge of designing the interconnection architecture for such hundred core MPSoCs. MPSoC implement a heterogeneous computation platform (consisting of programmable processors, application specific integrated circuits, re-configurable fabric) where each core supports a limited set of application domain functionality. For such designs, custom NoC architectures with optimized topologies have been shown to be superior to regular topologies (such as mesh or tori) in terms of power consumption and required NoC router resources. This work addresses the problem of synthesizing custom NoC architectures with the following considerations:

- Communication requirements: The communication requirements between the cores are typically specified by cumulative bandwidth (such as 10 Mbps). However, many MPSoC aimed at embedded domains implement streaming applications which demonstrate regular periodic transactions between the cores. Such communication patterns can be specified by transaction level (TL) specifications (described in the following section).
- Multiple use cases: In a current day high performance MPSoC only a subset of applications is active at any given time. The NoC synthesis approach should be able to effectively exploit the use case information to generate a topology that maximizes the resource sharing between the various use cases.

- IP library constraints: The NoC router and network to core interface library places constraints in terms of supported traffic classes, bandwidth constraints on ports, and arity constraints on routers.
- Deadlock avoidance: A key requirement of the synthesized NoC topology and routing scheme is that it must not result in deadlocks.
- Quality of results: The NoC synthesis approach must be able to effectively overcome the design complexity while generating solutions with guaranteed quality.

This chapter presents a holistic synthesis approach that is able to effectively address the above design requirements (communication, multiple use-cases and deadlock avoidance) and library constraints. Further, the approach is able to generate topologies with shortest path routes (demonstrating minimal latencies) and requiring minimum (optimum) dynamic power consumption, while consuming at most twice the number of router resources (and leakage power consumption) as the optimal solution.

2.2 Problem description

The inputs to the NoC design stage include the communication requirements of the cores in the MPSoC, the multiple use cases, floorplan information of the MPSoC computation architecture, and a library of (performance/power) characterized NoC IP blocks (routers and network interfaces).

The communication requirements for a MPSoC can be specified by graph $G(V, E)$ where $v \in V$ is the set of cores in the MPSoC and the set of directed edges $e(u, v) \in E$ denotes the communication requirements between

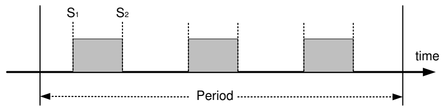


Figure 2.1: Transaction level specification

cores u and v. The edge e is either annotated with cumulative bandwidth requirement $\omega(e)$ (for example $\omega(e) = 10$ Mbps) between the cores or TL requirement $\lambda(e)$. The TL requirements are specified by $\lambda(e) = (p, L)$ where p is the period of the transaction specification and L is the list of transactions. Each transaction $l \in L$ is specified by a time range $l = [s_1, s_2]$ that denotes the potential start times for the transaction relative to the period. Figure 2.1 depicts the TL specification for an edge. Data dependent application behavior may cause the start time of the transaction to vary from one period to another and therefore we specify it as a range. Without loss of generality the size of all transactions (bit width and flit length) are assumed to be equal. The multiple use cases can be specified by a graph $G(V, E_1, E_2, ...E_n)$ where each set of edges E_i denotes the communication requirements for a particular use case.

We consider a simple NoC router architecture that only supports best effort traffic. The router components in the NoC library are characterized by the number of ports (η) and leakage power consumption, maximum arity or number of ports for any router in the library (η_{max}), maximum bandwidth supported on any one port (Ω), and the power consumed per unit bandwidth (ψ_r) to pass communication traffic through the router. The physical links are characterized by the power consumed per unit bandwidth of communication per unit length of the link (ψ_l).

The objective of the NoC synthesis technique is to generate an optimized topology with shortest path routes (consequently minimum dynamic power consumption) for each edge such that the communication latency is minimized. In addition to latency and dynamic power minimization, router resource reduction and thus leakage power minimization are also desirable objectives which are considered as secondary goals.

2.3 Related Work

Benini et al. [1] gives an excellent survey of the existing techniques for synthesizing custom NoC architectures. Existing approaches [2][7][8][3] only consider cumulative bandwidth requirements. Our approach is also able to consider transaction level specifications and exploit them for synthesizing NoC designs with low communication latency and jitter while using only best effort routers. Hansson et al. [4] and Murali et al. [5][6] proposed heuristic approaches for NoC synthesis with multiple application use cases. The communication requirements are specified by cumulative bandwidth requirements. The proposed approach that is based on an extension of Chatha et al. [2] approximation algorithm is able to synthesize NoC architectures for multiple use case while giving guarantees on quality bound. Further, the technique is also able to address both cumulative bandwidth and transaction level communication requirements for multiple use cases. Further, existing techniques [2][4] avoid deadlocks by including a post-synthesis step that introduces virtual channels at specific routers. While this approach is effective it does result in router IP modification which may not be desirable under all circumstances. This chapter presents an approach that is able to synthesize NoCs which do not contain deadlocks. Finally, in contrast to existing techniques, the proposed approach is able to address several NoC design requirements (cumulative bandwidth, transaction level specifications, multiple use cases, deadlock avoidance, router arity) in a holistic manner and give guarantees on design quality.

The proposed approach is based on a technique by Chatha et al. [2]. Their approach does not consider transaction level specifications, multiple use cases, bandwidth and port arity constraints, and deadlock avoidance. The bandwidth constraints and deadlock avoidance are addressed in a post NoC synthesis step in their approach. They are unable to incorporate port arity constraints in their basic approach (at least one variable greater than 0.5 no longer holds) and present an alternative strategy. Our approach is able to address all the shortcomings of Chatha et al. while utilizing their technique at the basic level. Thus, we are able to give near same quality guarantees as Chatha et al. while incorporating several additional requirements.

2.4 NoC Synthesis Technique

2.4.1 Design flow

We adapt the overall design flow proposed by Chatha et al. [2] for custom NoC synthesis (see Figure 2.2). The design flow begins by allocating routers at the corners of the channel intersection graph (CIG) of the MPSoC floorplan. The next step in the design flow is the core to router mapping stage. Chatha et al. [2] assumes that the cores are attached to one of the four routers located at its corner. They present an optimal approach (in terms of estimated dynamic power consumption and communication latency) for mapping the cores to the routers. We utilize their optimal core to router mapping algorithm. Figure 2.2

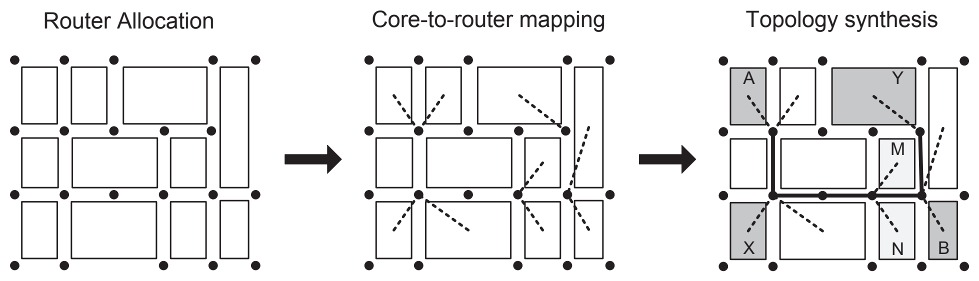


Figure 2.2: Overall design flow

depicts core to router mapping by dotted lines from the center of the core to the neighboring router. The final step in the NoC design flow adds the physical links between the various routers to construct the NoC topology and routes the communication transactions over the topology. The objective of the topology and router synthesis stage is to route each trace by minimum dynamic power consumption (thus also minimum latency) while utilizing minimum router resources (thus also leakage power consumption).

2.4.2 Basic approach to topology synthesis

Consider the topology synthesis stage shown in Figure 2.2. In the example we consider two communication traces between cores (A, B) and (X, Y), respectively. There are at least two potential shortest paths available for routing the traces. One which utilizes the routers on the top half and the other which utilizes the routers in the lower half of the layout. The synthesized topology utilizes the paths on the lower half of the layout as it requires fewer additional routers (note that the router connected to cores M and N is required to be in the NoC topology). Chatha et al. [2] present a polynomial time NoC synthesis algorithm that is able to route each trace by shortest path routes (optimum dynamic power consumption) while consuming at most twice the router resources as the optimal solution.

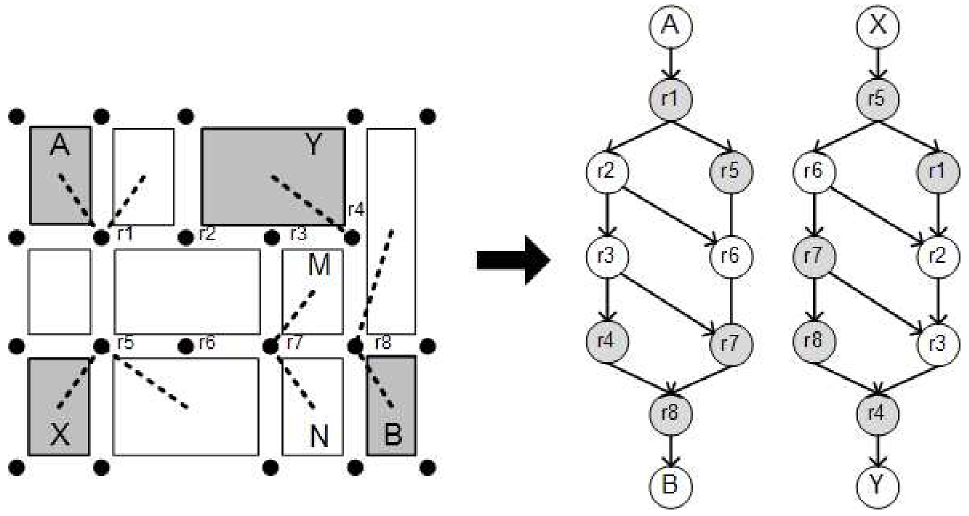


Figure 2.3: Shortest path graphs

Their approach relies on construction of shortest path graphs (SPG) as shown in Figure 2.3. The two graphs on the right show the SPG for cores (A, B) and (X, Y). The edges $(r2, r6)$ and $(r3, r7)$ denote alternative shortest paths created by over the cell routing of physical links. In the graph the routers that must be utilized (as cores are attached to them) in the synthesized NoC are shaded. Notice that the SPGs of the two cores share routers amongst them. Their technique minimizes the number of routers in the NoC subject to the constraint that a route exists from source to destination for each SPG. As the routes are selected from SPG they are all shortest path routes which minimize dynamic power consumption and communication latency.

The existing approach generates a SPG for every (u, v) in E. Essentially, the technique considers all possible shortest path routes for every communication requirement and generates a NoC design. It synthesizes a solution by utilizing a LP rounding based approach (see Figure 2.4). They prove that in each iteration the LP solution has at least one variable (that denotes a router

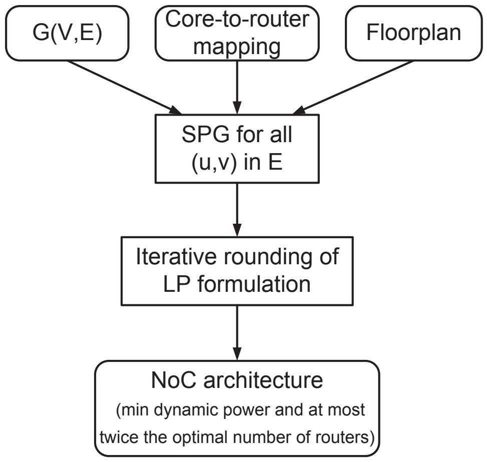

G(V,E)
Core-to-router mapping
Floorplan
SPG for all (u,v) in E
Iterative rounding of LP formulation
NoC architecture
(min dynamic power and at most twice the optimal number of routers)

Figure 2.4: NoC Topology Synthesis

is utilized in the NoC) above a 0.5 which is then rounded to 1. They also prove that such a rounding strategy generates a solution where the number of routers utilized in the NoC are at most twice the optimal. Thus, their approach converges to a solution in polynomial time with the above mentioned quality guarantees.

In the following sections we present extensions to SPG construction that can accommodate several NoC design requirements. Thus, we are able to give the same quality guarantees (shortest paths, minimum dynamic power, at most twice the optimal number of routers) on the solution as Chatha et al.

2.4.3 Deadlock avoidance

A deadlock is when no packets can progress further through the network or portion of the network. A deadlock is caused by routers forming a cycle and waiting on the resources of the next router in a cycle. This type of deadlock is typically referred to as a circular wait. Figure 2.5 illustrates an example of a deadlock. In the figure, the register(s) of each router are occupied with

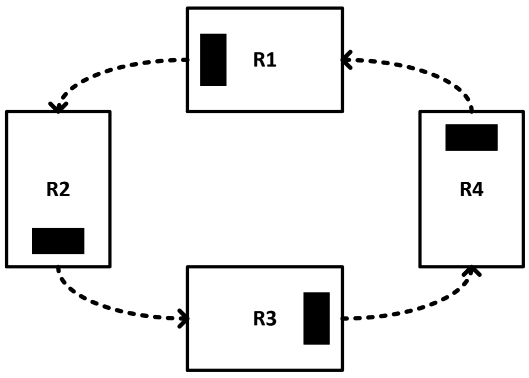


Figure 2.5: Example of a Deadlock

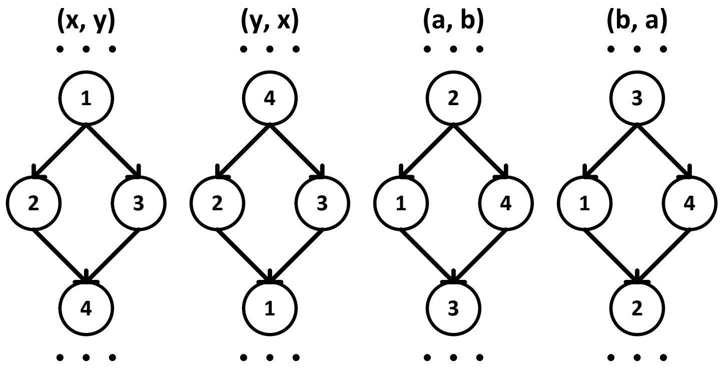


Figure 2.6: Deadlock Example - SPGs

contents attempting to be routed to the next router in the cycle. Since, the register(s) are filled no packets are able to transfer. Therefore, a deadlock has occurred.

Potential deadlocks can occur in the synthesized NoC if there are cycles in the channel dependency graph [9]. A channel dependency graph (CDG) can be obtained by transformation from the synthesized NoC. In the CDG the physical links of the NoC are denoted by nodes, and a directed edge between

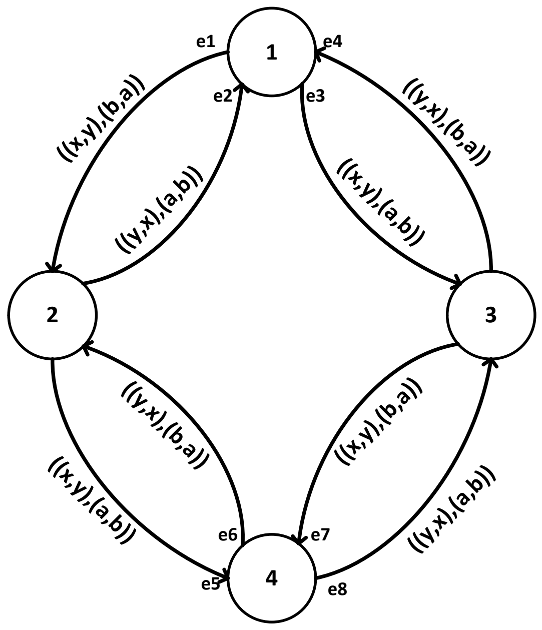


Figure 2.7: Deadlock Example - Global SPG

two nodes $(u, v) \in CDG$ denotes that a communication requirement is routed from node u to v (both u and v represent physical links in the NoC). Deadlocks in a NoC can be alleviated by introduction of additional virtual channels (or buffers) in the routers, and breaking the cycles [9]. However, such an approach does lead to modification of the routers. An alternative strategy that is presented here would be to generate a NoC that does not have cycles in its CDG.

The approach is based on the modification of the *SPGs*. We first find the shortest path graphs (SPGs) for each communication trace in the design. A SPG represents all of the shortest routes (# of routers) for a communication trace. As an example, Figure 2.6 illustrates four communication traces and their corresponding SPGs. After each of the individual shortest path graphs have been found, all of the SPGs are combined into a globel shortest path

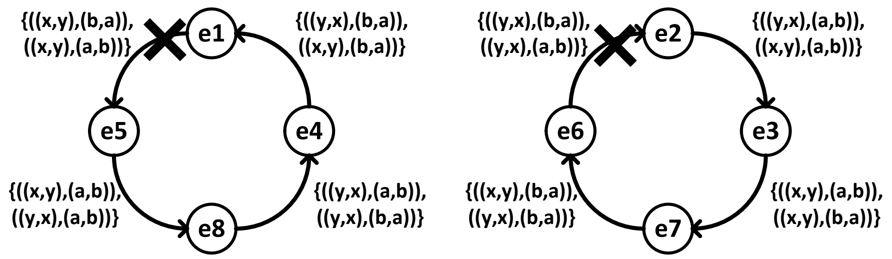


Figure 2.8: Deadlock Example - Global CDG

graph (SPG_G). The global shortest path for our example is illustrated in Figure 2.7. Each edge in the SPG_G is annotated with the traces that are routed along the edges.

After the SPG_G is generated the graph is transformed into a CDG_G. A CDG_G is a global Channel Dependency Graph (CDG). In the CDG_G a vertex represents an edge in the SPG_G. An edge $(e_1, e_2) \in CDG_G$ if and only if a path is present in the SPG_G that contains edges e_1 and e_2. Figure 2.8 illustrates the CDG_G for the SPG_G. Each edge in the figure is annotated with the traces that have paths through the edges. After the CDG_G is generated we find all of the strongly connected components (SCCs). In a SCC there is a path from each node to every other node in the SCC. Therefore, a SCC in the CDG_G represents a potential cycle in the network and thus a potential deadlock. By removing an edge(s) from the SCCs we will break all possible cycles and therefore ensure that no deadlocks are possible. In the CDG_G in Figure 2.8, there are two SCCs as easily seen in the figure. If we remove an edge from each of the SCCs we will break the cycles. For instance, assume we eliminate the edge between nodes e_1 and e_5 and the edge between nodes e_2 and e_6. After the edges have been removed the changes are reflected back into the individual SPGs.

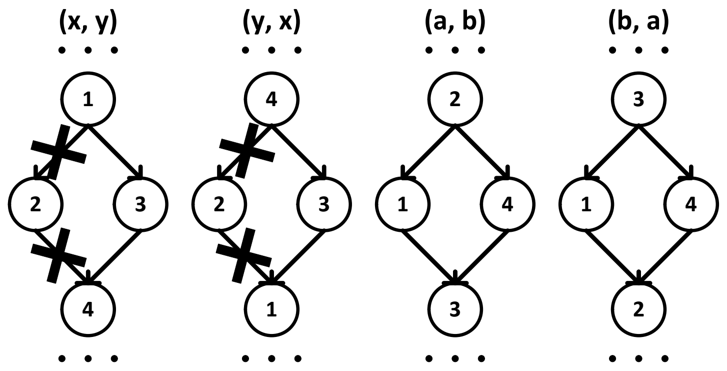


Figure 2.9: Deadlock Example - Updated SPGs

Figure 2.9 illustrates how the removal of the edges from the CDG_G is reflected back onto the SPGs. In the figure, trace (x, y) has had the edges from node 1 to node 2 to node 4 removed. This is from the edge in the CDG_G from node e_1 to e_5 being eliminated. Addionally, in Figure 2.9 the trace (y, x) has had the edges from node 4 to node 2 to node 1 removed. This is from the edge in the CDG_G from node e_2 to e_6 being eliminated. With these changes to the SPGs there is no longer any risk of circular wait deadlocks in the final synthesized network.

The pseudo code for modifying $SPGs$ for deadlock avoidance shown in Figure 2.10. The above discussion addressed the problem in the context of a single use case. The same identical steps can be followed for multiple use cases to avoid deadlocks in each of them. The for loop in line 2 iterates through all the use cases. Line 3 and 4 generate the SPG_G and CDG_G structures, respectively. Line 5 finds all the SCC in CDG_G, and the for loop of Line 6 removes back edges in each of them to eliminate cycles. Finally, Line 10 reflects the changes on the individual $SPGs$ by removing appropriate edges.

```
$deadlock\_avoidance()$
for each use-case $UC$ do
    $generate\_global\_SPG()$
    $convert\_SPG\_to\_CDG()$
    $find\_SCCs()$
    for each $SCC$ do
        $find\_back\_edges()$
        $remove\_back\_edges()$
    end for
    $update\_SPGs()$
end for
```

Figure 2.10: Pseudocode for deadlock avoidance

The complexity of *deadlock_avoidance*() is dominated by Line 3 that generates SPG_G. Let U denote the number of use cases in the application, and let R denote the number of routers allocated to the floorplan. We utilize Dijkstra's shortest path algorithm to generate the SPG for a single trace which has a complexity of $O(R^2)$. SPG_G construction in Line 3 has a complexity of $O(TR^2)$ where T denotes the maximum number of traces over all use cases of the application. The overall complexity of Line 3 is $O(UTR^2)$ when the for loop of Line 2 is taken into account.

2.4.4 Communication requirements

Cumulative bandwidth requirements: We first consider cumulative bandwidth requirements and then discuss transaction level specifications. The basic SPG construction (as described in Chatha et al.) assumes that when routes are shared between two communication traces (in order to minimize router resources) there is no violation of port bandwidth constraints. However, violations could exist and we address them by splitting the traces across differ-

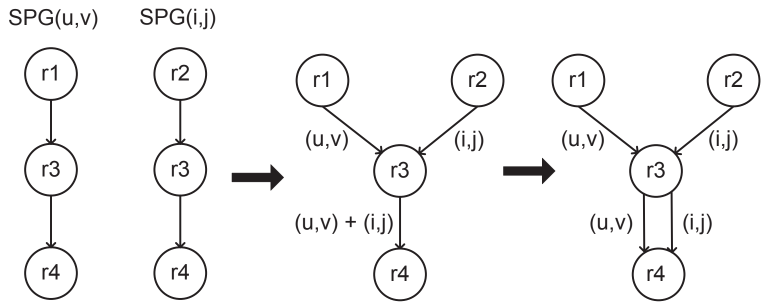


Figure 2.11: Bandwidth violation

ent router ports. Figure 2.11 depicts partial SPGs of two traces (u, v) and (i, j) that could potentially be routed through routers $r3$ and $r4$. However, $\omega(u, v) + \omega(i, j) > \Omega$. We avoid the bandwidth violation by routing (u, v) and (i, j) on different ports of $r3$ and $r4$ as shown in the figure. Thus, if router $r3$ and $r4$ were to be utilized for routing (u, v) and (i, j) in the final design, they would be routed on different ports of $r3$ and $r4$. We incorporate this information in the SPG by annotating it on router $r3$.

Transaction level specifications: Inclusion of transaction level specifications in the communication requirements gives us an opportunity to avoid conflicts or interference between the various transactions. We capture transaction interference by first traversing each transaction from source router (connected to the initiating core) to the sink router (connected to the target core) along each shortest path. As the transaction traverses a router the range of its start time is delayed by the router switching delay. Thus, we know the range of start times of each transaction at every router along all its shortest paths.

We consider three cases for interference between two transactions that wish to access the same output port of a router (see Figure 2.12). The rectangles for transactions 1 and 2 denote the range of their start times. The shaded

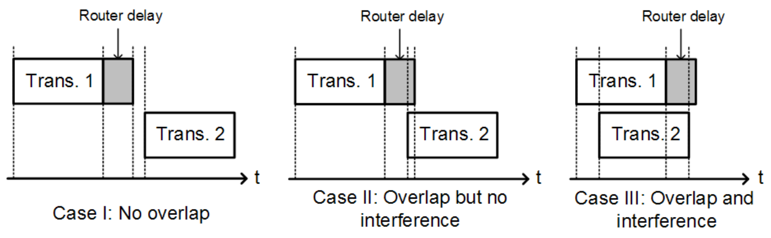


Figure 2.12: Transaction interference

box represents the minimum transmission delay for a transaction through a router. In Case I transactions 1 and 2 do not interfere with each other as there is no overlap between their ranges of start times. In Case II if transaction 1 starts at its latest possible time, and transaction 2 starts at it earliest possible time, then the two transactions overlap as transaction 1 is being switched through the network. However, we permit such overlaps to happen. Finally, in Case III if transaction 1 starts at its latest possible time, transaction 2 could delay transaction 1 if it starts earlier. We treat the latest start time of the transaction plus the router switching delay as the deadline for the transaction. Hence, Case III could potentially cause a deadline violation on transaction 1, and we avoid such scenarios.

We avoid interference between two transactions (Case III) by adopting a similar approach as that for bandwidth violation. The two transactions are assigned to two different ports that are connected with the same neighboring router. Thus, both the transactions can traverse to the same neighboring router without interfering with each other.

Cumulative and transaction specifications: Finally, we also avoid conflicts between communication requirements that are specified by cumulative bandwidth and those that are specified by transaction specifications. We do permit

```
 1: $trace\_conflict\_resolution()$
 2: for each router $R$ do
 3:    for each use-case $UC$ do
 4:       for each trace $T1$ placed on router $R$ do
 5:          for each trace $T2$ placed on router $R$ do
 6:             if $T1 \neq T2$ then
 7:                if $conflicts\_exist(T1, T2)$ then
 8:                   $resolve\_conflict(R, T1, T2)$
 9:                end if
10:             end if
11:          end for
12:       end for
13:    end for
14: end for
```

Figure 2.13: Pseudocode for trace conflict resolution

cumulative bandwidth and transaction specifications originating from the same core to share the same route.

The pseudocode for the trace conflict resolution algorithm is shown in Figure 2.13. The algorithm iterates through each router, R, in the topology (line 2) for each use case, UC (line 3). The algorithm then compares each trace, $T1$, on the router R with every other trace, $T2$, (lines 4-7) also on the router. The two traces are compared to see if they are routed through the same port, and if they are whether a conflict exists (line 7). If a conflict is found between the two traces the conflict is resolved as discussed above (line 8). The complexity of $trace\_conflict\_resolution()$ algorithm is determined by the for loops of Lines 2, 3, 4 and 5, and it is $O(RUT^2)$.

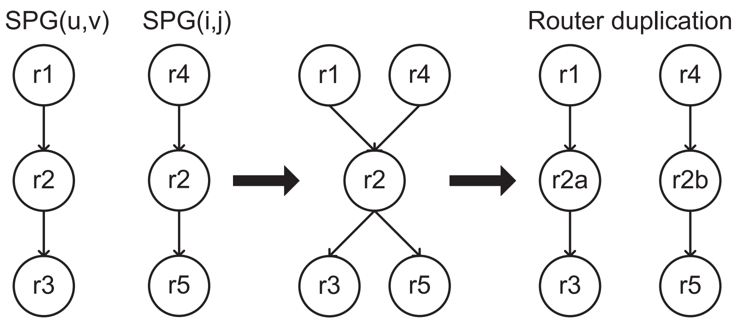


Figure 2.14: Router arity constraints

The addition of ports to avoid bandwidth violations and transaction interferences can potentially lead to router arity constraint violations whose alleviation is discussed in the next section.

2.4.5 Port arity constraints

Figure 2.14 depicts port arity constraint alleviation. The left hand side of the figure shows partial SPGs belonging to communication requirements between cores (u, v) and (i, j). We assume the port arity constraint is 3. If in the final NoC topology both the communication requirements are routed through $r2$ they would cause a port arity constraint violation. We avoid such a violation by introducing a duplicate router at the same location as $r2$. Thus, there are now two routers $r2a$ and $r2b$ located very close to each other. The SPGs are modified to utilize $r2a$ for (u, v) and $r2b$ for (i, j).

The pseudocode for router port arity resolution algorithm is shown in Figure 2.15. The algorithm begins by iterating through each router, R, and each use-case, UC (lines 2 and 3). Each router R has its port arity compared with MAX_PORTS which denotes the port arity constraint (line 4). If router R has too many ports the algorithm finds the set of routers, Z, connected to

```
 1: $port\_arity\_resolution()$
 2: for each router $R$ do
 3:    for each use-case $UC$ do
 4:       if $port\_arity(R) > MAX\_PORTS$ then
 5:          $Z \leftarrow$ set of routers connected to router $R$
 6:          $T \leftarrow$ set of traces passing through router $R$
 7:          for each combination $K_i \in C^{Z}_{MAX\_PORTS}$ do
 8:             $S = duplicated\_router(R)$
 9:             $connect\_routers(S, K_i)$
10:             for each trace $t$ in $T$ do
11:                if $t$ has a path through $R$ using routers in $K_i$ then
12:                   $add\_edges()$
13:                end if
14:             end for
15:          end for
16:          $remove\_router(R)$
17:       end if
18:    end for
19: end for
```

Figure 2.15: Pseudocode for router port arity resolution

router R as well as the set of traces, T, passing through the router (lines 5 and 6). For each combination $K_i \in C^{Z}_{MAX\_PORTS}$ of routers from Z, the algorithm creates a new router S by duplicating the router R (lines 7 and 8). Next, the K_i combination of routers are connected to the new router (line 9). The algorithm then iterates through each trace t in the set of traces T (line 10). If trace t has a path through router R using the routers in K_i, the algorithm introduces edges between S and appropriate routers in K_i (lines 11 and 12). After each combination of routers from Z have been processed the algorithm removes the initial router R (line 16). The algorithm terminates after each use-case has had all of its routers processed.

The complexity of $port\_arity\_resolution()$ algorithm is dominated by for loop of Line 7 which has an overall complexity of $O(TR^{MAX\_PORTS+2})$. The overall complexity of the algorithm is $O(TUR^{MAX\_PORTS+3})$ or $O(TUR^8)$ if MAX_PORTS is assumed to be equal to 5.

2.4.6 Multiple use cases

The basic approach [2] to topology synthesis (Section 2.4.2) creates a set of SPGs (each SPG is associated with one $(u, v) \in E$) from $G(V, E)$. A set of SPGs associated with an edge set E is denoted by $SPG(E)$. We can address multiple use cases by creating multiple sets of distinct sets of $SPG(E_i)$ associated with each use case $E_i \in G(V, E_1, \ldots, E_i)$. The bandwidth and transaction interference constraints imposed by E_i are only addressed in individual $SPG(E_i)$. The bandwidth, transaction interference and deadlock avoidance constraints are only applied on individual $SPG(E_i)$. The port arity constraints are imposed on all $SPG(E_i)$. All the $SPG(E_i)$ are then input to the LP rounding based technique. Consequently, the resulting solution has shortest routes for each communication requirement, and minimizes the number of routers across all use cases. Thus, the approach is able to effectively minimize the resource usage across all use cases.

2.4.7 Algorithm Time Complexity Analysis

The algorithm time complexity for constructing the $SPGs$ utilized by our approach is dominated by the algorithm, $port\_arity\_resolution()$, which is $O(TUR^8)$ (if MAX_PORTS is assumed to be equal to 5). The iterative NoC synthesis step utilizes approach identical to Chatha et al. which is polynomial. Thus, the overall time complexity of can be considered to be $O(TUR^8)$.

Table 2.1: Categories of synthetic benchmarks

	Bandwidth specification		
	Cumulative	Transaction	Mixed
Multiple use case	S1	S2	S3

2.5 Experimental Results

2.5.1 Experimentation set-up

Benchmark designs We performed extensive experimentation to evaluate our technique. We considered several categories of synthetic benchmarks and a realistic benchmark application. A description of the various categories of synthetic benchmarks is given in Table 2.1. All the synthetic benchmark categories considered multiple use cases and had 10 designs within each category. The synthetic benchmarks in $S1$ were generated by randomizing the number of cores (ranging from 10 to 50 cores), number of traces, bandwidth per trace and number of use cases. In the case of benchmarks in category $S2$, the randomization on bandwidth was replaced by randomization on the period of transactions associated with a trace (instead of bandwidth), number of transactions within the period, amount of data in each transaction, and the start window of the transactions. Finally, synthetic benchmarks in category $S3$ were generated by an additional randomization on type of a trace (cumulative versus transaction).

Our realistic benchmark modeled a real-world multimedia application comprising of video recording, video playback, and wireless communication. The application consisted of 21 cores broken into memories, processing units, as well as application specific cores. The benchmark consisted of three applications and three use-cases: a use-case with only video playback and wireless communication, a use-case with video recording and wireless communication, and a use-case with video playback and recording and wireless communication. Each application of the benchmark was given as a transaction level specification.

In our experiments with both synthetic and benchmark traffic we considered a port arity constraint of 5. We also present experimental results that evaluate our approach by varying the port constraints from 3 to 10.

Existing approaches We compared the technique against two existing approaches: an integer linear programming approach (referred to as "ILP" for the remainder of the chapter) by Srinivasan et al. [3] and a heuristic synthesis technique rhat generates NoC with guaranteed throughput router architectures by Hansson et al. [4] (hence forth referred to as "GT"). Both these approaches consider only cumulative bandwidth specifications and do not support transaction level traffic specifications. The ILP synthesizes designs with best effort routers, and therefore is a good representative of many other existing NoC design approaches. Similar to the other approaches ILP also does not consider multiple use cases, port arity constraints, and deadlock avoidance. ILP addresses multiple use cases by generating a NoC architecture for the worst case scenario. The worst case scenario is one in which all the applications are active simultaneously. We compared our approach with ILP for synthetic benchmarks that include cumulative bandwidth traffic ($S1$ and $S3$) and the realistic application.

The GT approach heuristically searches for shortest paths to route a trace subject to bandwidth constraints on the intermediate router. The

GT approach uses a specialized router architecture that assigns guaranteed bandwidth to each trace flowing through the router. Thus, GT is a good representative of existing NoC synthesis approaches that consider guaranteed throughput traffic. The bandwidth guarantees are achieved by including a table at each port in the router. The base GT approach is able to handle multiple use cases, but does not consider port arity constraints. We extended the base GT approach [4] to account for port arity constraints. We compared our approach with GT for synthetic benchmarks that include transaction level traffic specification ($S2$ and $S3$) and the realistic application.

Evaluation metrics We compared against the existing approaches by evaluating the power consumption, router requirements, and performance of the generated designs. The power consumption included both dynamic (due to routers and physical links) and leakage (due to routers) power for the designs. The power consumption numbers were generated through RTL synthesis using a 65nm lower power process. The average packet latency was determined through simulation using a transaction level simulator. The simulations included a warm up period before the performance data was recorded. In the case of designs that included transaction-level traffic information, the transactions were launched uniformly at random within their start time window. The cumulative bandwidth traffic was also generated by launching flits uniformly random within a specified period (reciprocal of bandwidth). The simulations recorded the latency of each transaction along with the number of hops it traversed as it traveled through the network. Thus, we could calculate the average latency and standard deviation for every packet hop distance. In all our charts that compare the average latency of the various approaches we also

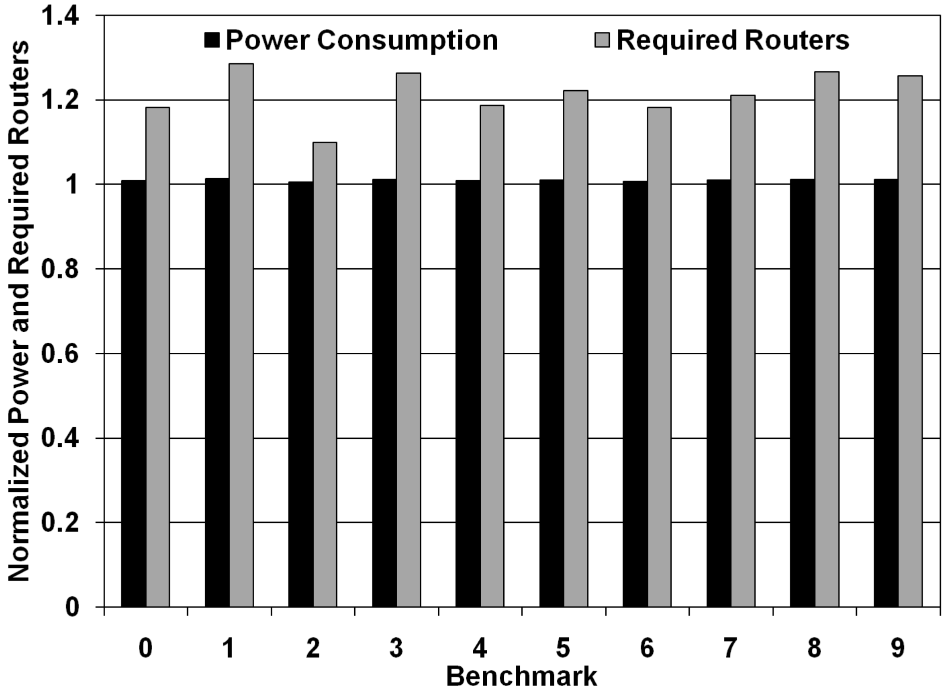


Figure 2.16: Power/Router for ILP

plot the lower bound of the latency. The lower bound is calculated as the minimum latency for a flit to traverse the specified number of hops with no interference from any other trace.

We also performed additional experiments that evaluated the quality of the designs as the bandwidth was increased, port arity constraint was varied (only for our approach), allowable transaction interference was increased (only for our approach), and an application with deadlock possibility was considered.

2.5.2 Comparisons with existing approaches

Comparison with ILP for $S1$ We begin the discussion by presenting results that compare our approach with ILP for the synthetic benchmark category $S1$. Figure 2.16 plots the normalized power consumption and router require-

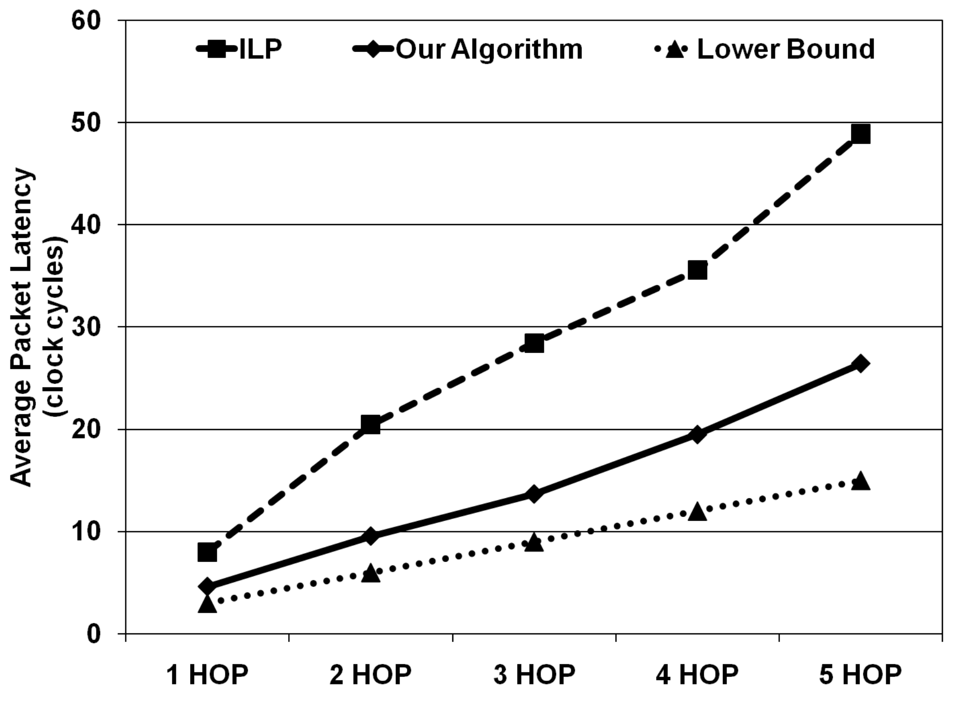


Figure 2.17: Avg. packet latency

ments for the NoC designs generated by ILP technique with respect to our approach. As can be seen from the plot the power consumption of the designs are comparable while ILP designs require a lot more routers. It is to be expected as the ILP approach does not optimize for multiple use cases and instead generates a NoC design for the worst case scenario. Further, we would also like to point out the ILP approach did not honor port arity constraints in any of the synthesized NoC designs. Figures 2.17 and 2.18 plot the hop wise latency and standard deviation, respectively of the designs generated by ILP and our approach. Figure 2.17 includes a plot for the lower bound on packet latency for each hop assuming no interference. It can be seen from the plots the designs generated by our technique demonstrate much lower packet latencies and standard deviation with respect to the designs synthesized by

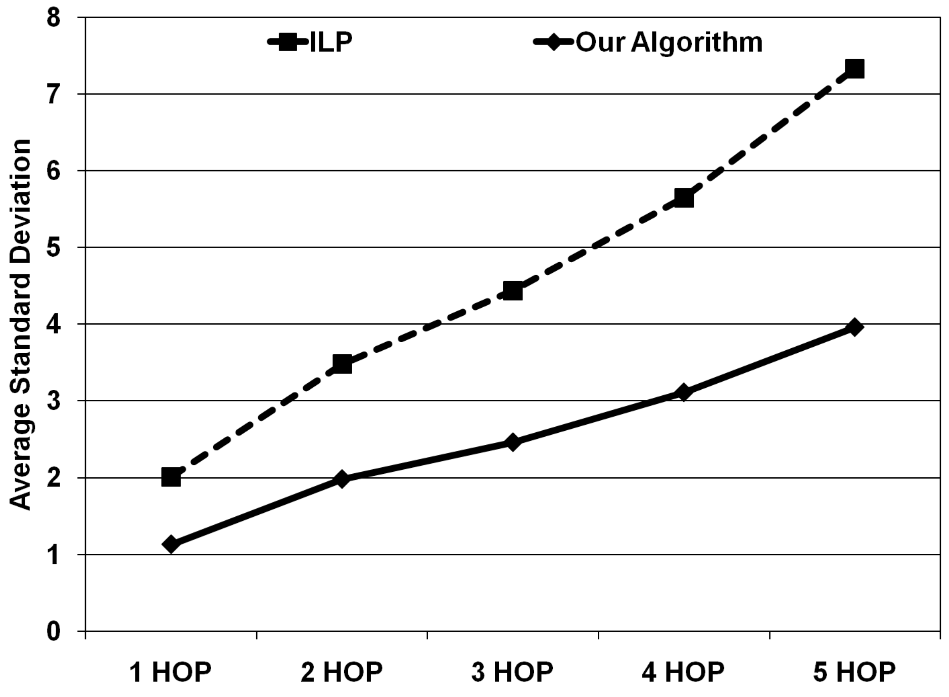


Figure 2.18: Avg. standard deviation

ILP. The designs synthesized by ILP do not honor port arity constraints and utilize routers with very large number of ports. Consequently, there is a lot of interference between the various traffic traces that traverse such a large port arity router leading to higher average packet latency and standard deviation.

Comparison with GT for $S2$ We next present experimental results that compare our approach against GT with synthetic benchmarks in category $S2$. Figure 2.19 plots the normalized power consumption and router requirements for the designs generated by GT in comparison to our approach. It can be observed from the figure that the GT designs utilize higher power in all cases. However, the router resource requirement is lower for the GT approach. The higher power consumption requirement is attributed to the complex router

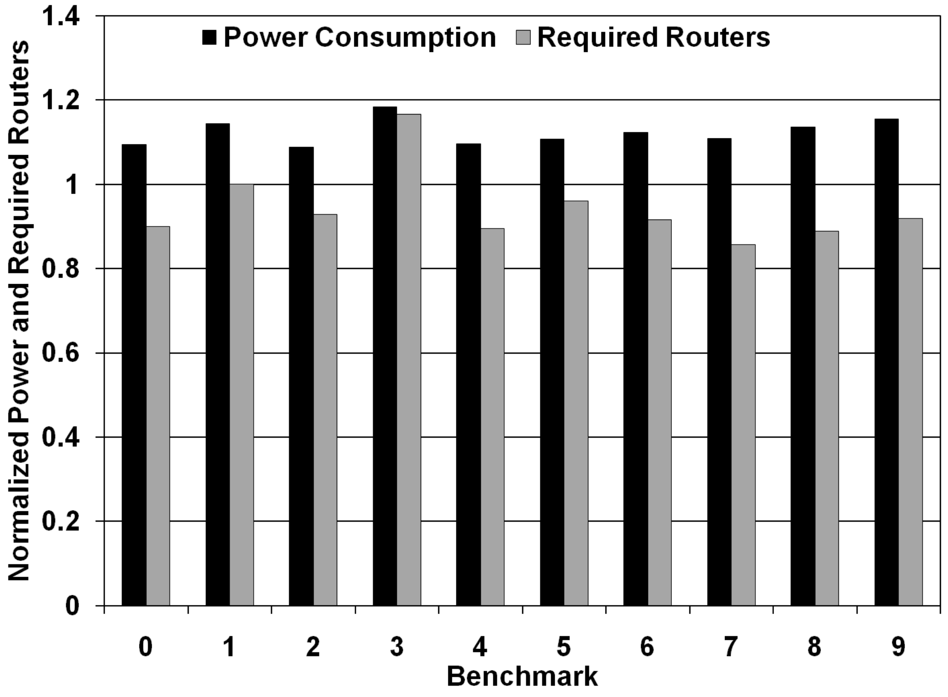


Figure 2.19: Power/Router for GT

architecture utilized by the GT approach. In contrast our technique utilizes simple best effort routers. The higher router requirement of our approach is due to the transaction conflict avoidance measures taken by our technique. Our approach utilizes more routers in an effort to generate alternative paths in the NoC that lower conflict between various transactions. Figures 2.20 and 2.21 plot the per hop latency and standard deviation, respectively for the designs synthesized by GT and our approach. As can be seen from the figures, our designs demonstrate lower average latency and standard deviation with respect to the GT designs. This is significant as our designs are based on simple best effort routers as opposed to the GT designs that utilize complex guaranteed throughput designs. Therefore, although our technique generates designs that require more routers, the designs demonstrate lower power consumption and superior performance.

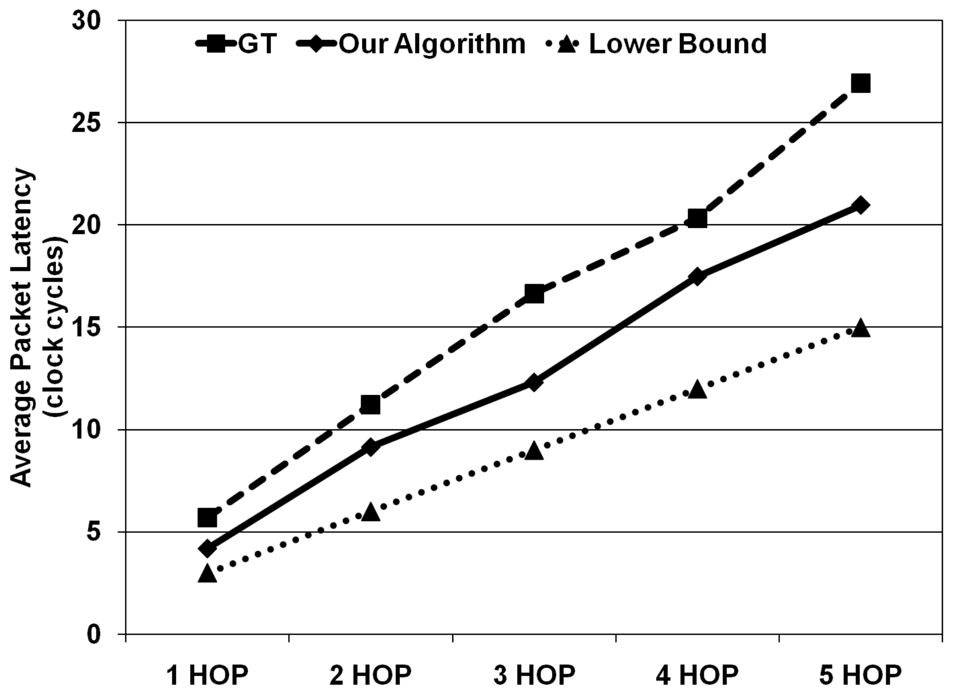


Figure 2.20: Avg. packet latency

Summary of comparisons with ILP and GT In this section we summarize the comparisons with ILP and GT approaches for the synthetic benchmarks and the realistic application. Figure 2.22 gives the average percentage reduction in routers and power consumption due to our technique in comparison with both ILP and GT. The X-axis of the plot depicts the improvements for the three synthetic benchmark categories and the realistic application. We compared only against ILP for category $S1$, and category $S2$ was used for comparison only with GT. Figure 2.23 gives the average percentage reduction in latency and standard deviation due to our approach. The conclusions of the study are summarized below:

1. Category $S1$: The ILP generated designs that required comparable power consumption, almost 20% additional routers and that demonstrated about

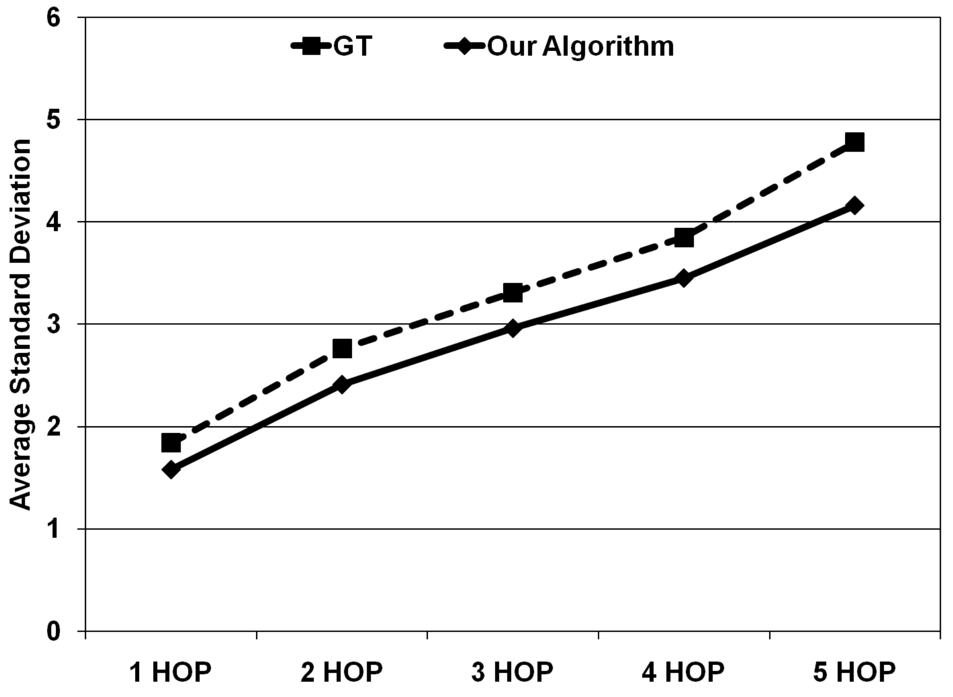


Figure 2.21: Avg. standard deviation

45% higher average latency and standard deviation. The ILP performed poorly because it does not optimize for multiple use cases, and does not consider port arity constraints.

2. Category $S2$: The GT generated designs that required over 10% additional power consumption, 5% lower router resources and whose average latency and standard deviation were higher by over 20% and 10%, respectively. This result is significant as our technique is able to generate high quality designs while using best effort routers as opposed to GT that utilized higher complexity guaranteed throughput routers.

3. Category $S3$: Synthetic benchmarks in this category modeled applications with both mixed cumulative and transaction traffic specifications. Designs synthesized by our approach demonstrated superior performance

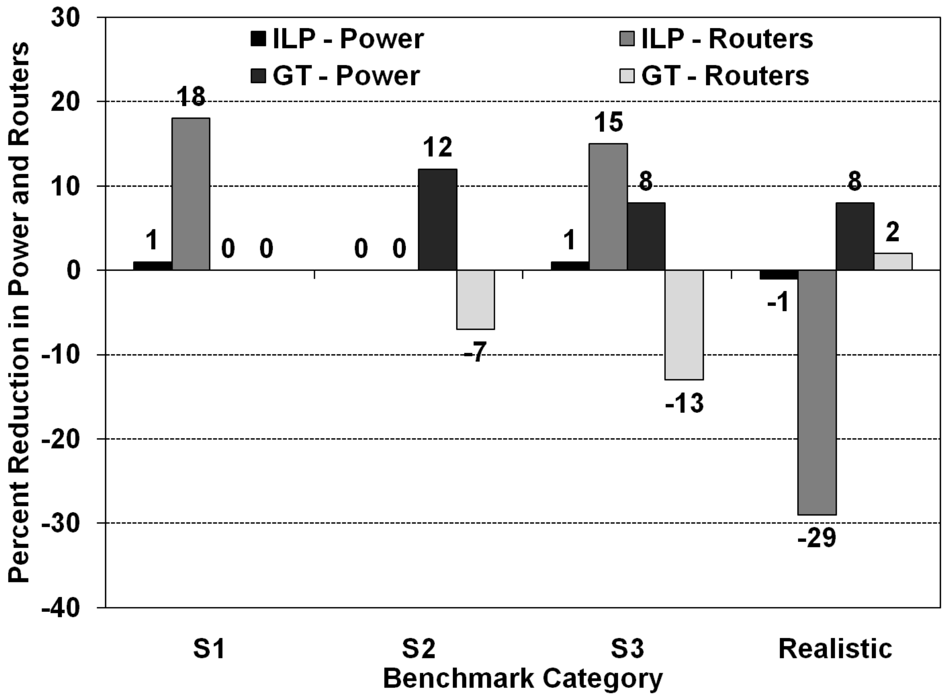


Figure 2.22: Power/router comparisons

in comparison to the solutions of ILP and GT approaches. The trends in power consumption and router resource requirements observed for the previous two categories are repeated for category $S3$ for precisely the same reasons.

4. Realistic application: The realistic application consisted of transaction level traffic specifications. Designs generated by ILP do not consider interference avoidance between transactions and therefore require fewer router resources. The fewer router resources also result in lower power consumption of the ILP design because of reduced leakage power consumption. However, the average latency and standard deviation in latency is very large for ILP design. The GT approach generates a design that requires higher power consumption and comparable router

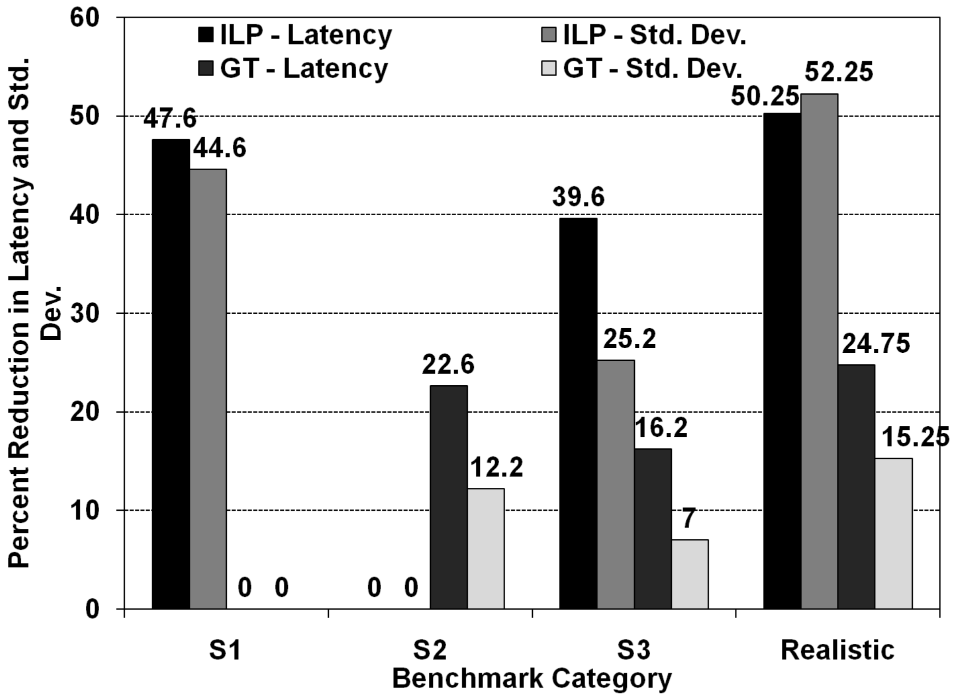


Figure 2.23: Latency/std. dev. comparisons

resources. The higher power consumption is primarily because of the complexity of the guaranteed throughput router architecture. The performance of the GT design showed a similar trend as that observed with $S2$ synthetic benchmarks.

The run times of the various approaches were compared for large 50 core synthetic benchmarks. The ILP approach had to be timed out at 12 hours to generate the solutions. The GT approach took 5 minutes on average while our approach took 45 minutes. It must be noted that our approach is able to give very tight bounds on the quality of the solutions (shortest path routes, minimum dynamic power consumption, at most twice the number of routers and leakage power consumption as optimal solution) while addressing multiple design requirements (multiple use cases, cumulative and transaction traffic specification, port arity constraints and deadlock avoidance).

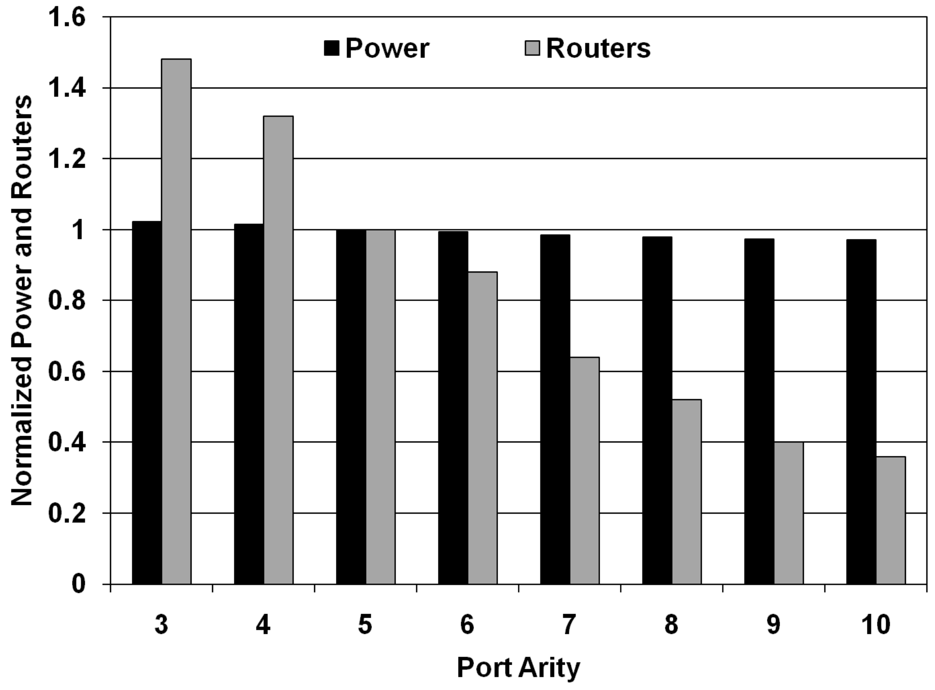


Figure 2.24: Power/Routers versus Port Arity

2.5.3 Impact of port arity

We examined the impact of maximum port arity on the power consumption and required router resources for our technique. We utilized the set-top box benchmark from Srinivasan et al. [3]. We first generated solutions for the benchmark using our technique with the default maximum port arity setting of 5 ports. We then varied the maximum port arity from 3 ports to 10 ports and generated solutions for each constraint. The results for power consumption and required router resources are shown in Figure 2.24. The plotted values have been normalized to the solution given by our technique for a maximum port arity of 5 ports. Increasing the port arity reduces both the power consumption and required router resources. The reduction in router resources is

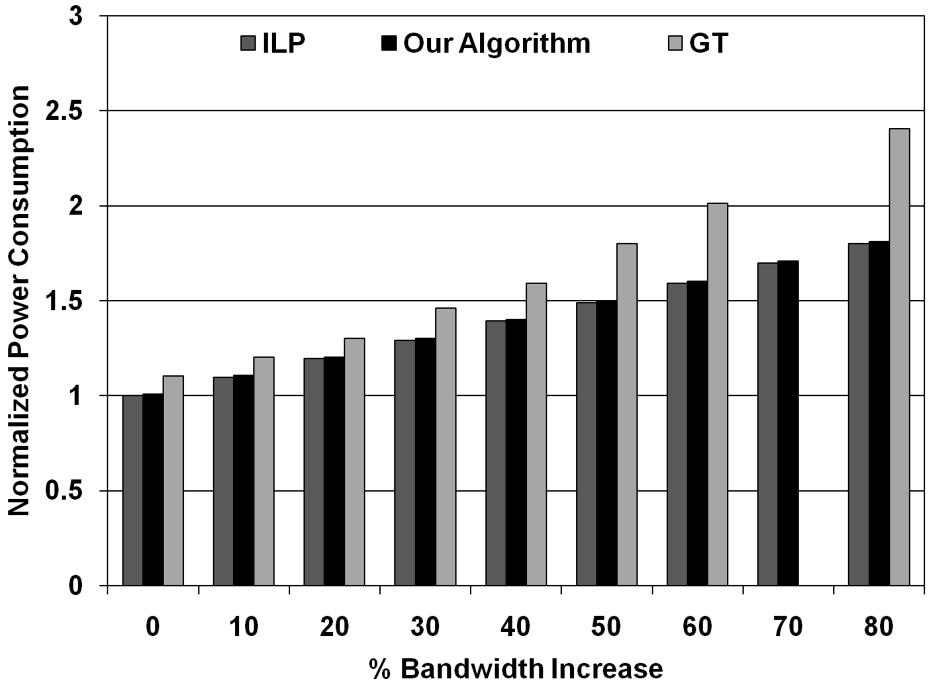


Figure 2.25: Power Consumption with Increasing Bandwidth

quite substantial. The reduced router resources are also responsible for the nominal decrease in power consumption due to lower leakage power consumption.

2.5.4 Impact of increasing bandwidth

We also studied the impact of increasing the bandwidth of an application on the generated NoC architecture. We consider a synthetic single use case benchmark consisting of 14 cores, and 45 traces (specified at transaction level). The total bandwidth flowing through the traces was 31000 Mbps (with 13 Mbps and 720 Mbps as minimum and maximum trace bandwidth, respectively). The supported bandwidth at a router port was 4200 Mbps. We increased the bandwidth of each trace in steps of 10% of initial bandwidth till the bandwidth of a

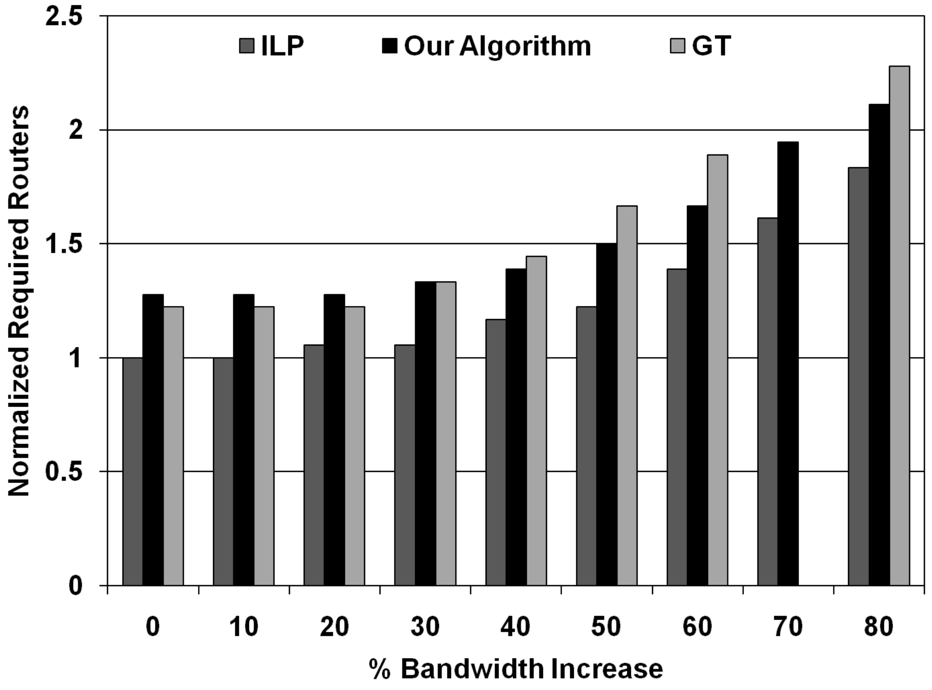


Figure 2.26: Required Routers with Increasing Bandwidth

trace was doubled (100% increase). We compared the NoC designs generated by our technique against the ILP and GT solutions.

Figures 2.25 and 2.26 plot the power consumption and required routers for the various designs normalized to the solution generated by ILP for original design (0% bandwidth increase). It can be observed that as the bandwidth is increased the power consumption of the designs increases. The power consumption of the solutions generated by our approach are comparable to those generated by ILP while the GT designs utilize considerably higher power. Further, the power consumption of the GT designs increases at a faster rate with increase in bandwidth in comparison to solutions of ILP and our approach.

The router requirements of the designs remain more or less constant for bandwidth increases of up to 30%. Beyond 30% the number of routers begin

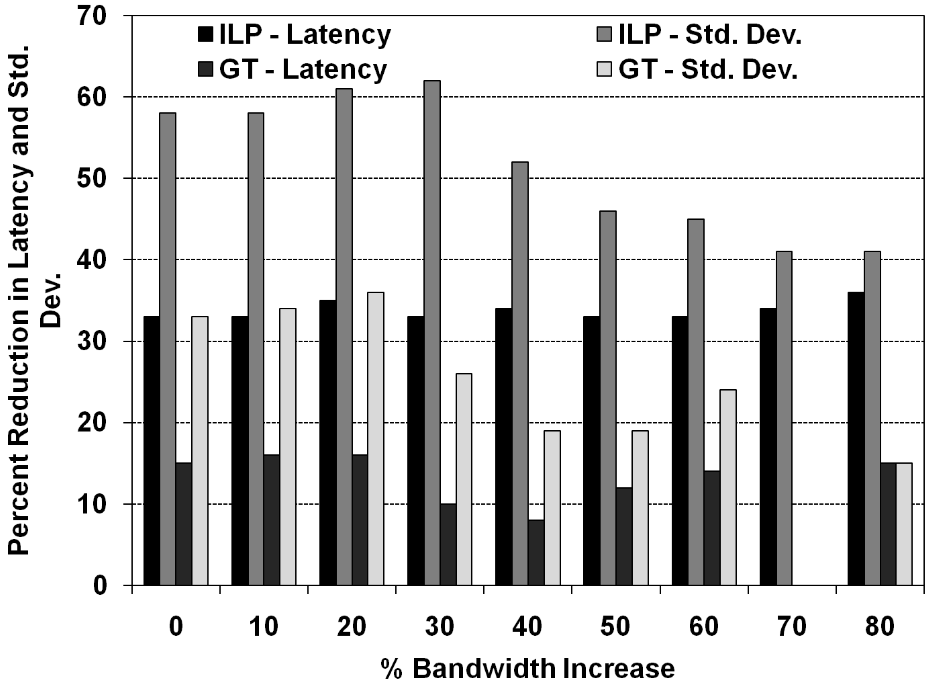


Figure 2.27: Reduction in latency and standard deviation

to increase with bandwidth for the solutions of the three approaches. The ILP solutions utilize lower router resources than GT and our approach. At lower bandwidth requirements the router requirements of GT are comparable to our approach. However, at higher bandwidth requirements GT solutions utilize markedly higher router resources. The GT approach failed to generate solutions for inputs with 70%, 90% and 100% increase in bandwidth. The ILP approach failed for inputs with 90% and 100% bandwidth increases. Both the GT and ILP approaches consider an initial allocation of routers, and if they are unable to generate the NoC design with the initial allocation they declare failure. Our approach replicates routers as required, and therefore was able to generate valid solutions for all inputs.

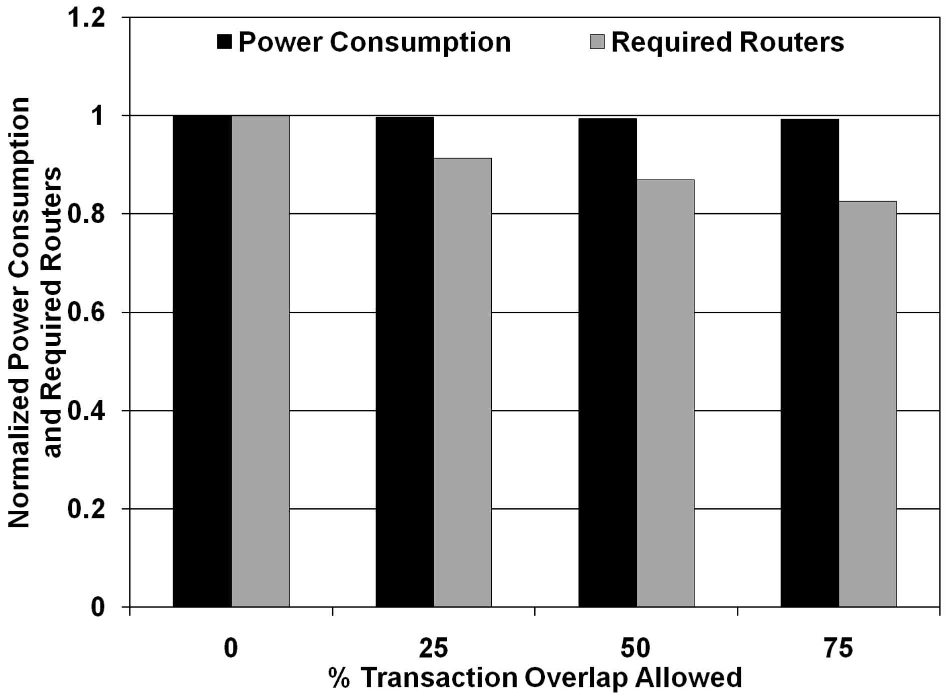


Figure 2.28: Normalized Power and Routers with Increasing Transaction Overlap

Figure 2.27 plots the percentage reduction in average latency and standard deviation due to our approach in comparison with ILP and GT solutions. The percentage reductions in both average latency and standard deviation are higher for ILP than GT solutions. The average latency reductions remain more or less constant even as the bandwidth is increased. Although, the standard deviation reductions reduce slightly as the bandwidth is increased, they are still quite large (40% for ILP and 15% for GT) for NoC designs with 80% higher bandwidth.

2.5.5 Degree of transaction interference

We also analyzed the impact on the quality of the designs as the degree of transaction interference was varied. The degree of transaction interference

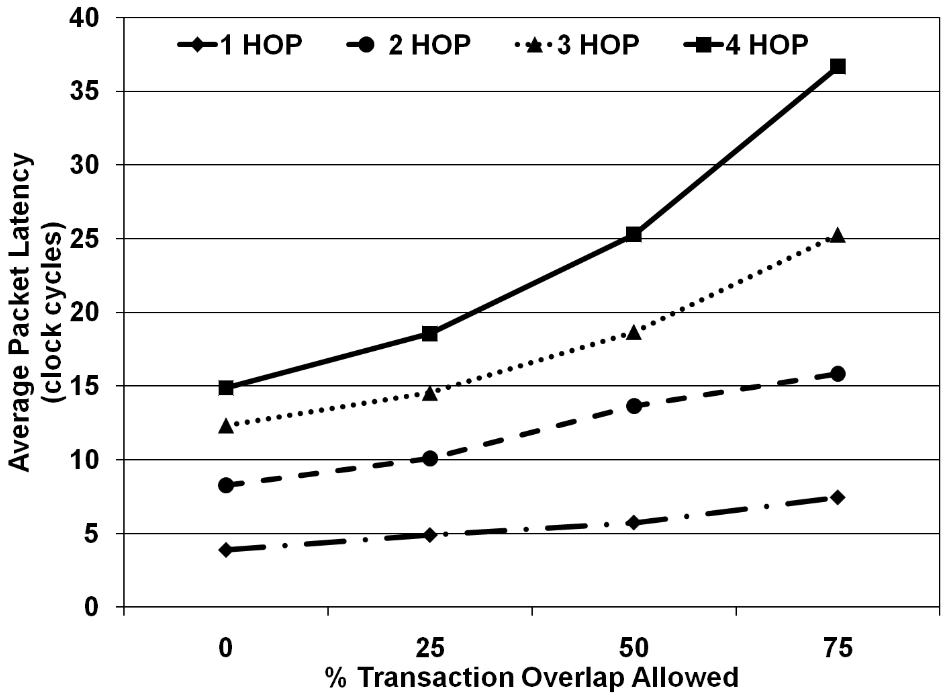


Figure 2.29: Average Packet Latency with Increasing Transaction Overlap

is specified as the percentage of overlap between the start time windows of two transactions. The percentage overlap is measured with respect to the transaction that has the smaller start time window. We considered the same benchmark as the previous example. We generated designs with 0%, 25%, 50% and 75% permitted overlaps between the transactions. Figure 2.28 plots the power consumption and router requirements of the 4 designs normalized to the 0% overlap design. As observed from the figure, the router requirement reduces with increase in overlap. Our approach avoids transaction interference by introducing additional routers and constructing alternative routes. Thus, an increase in permitted interference leads to a decrease in router resources. The router resource reduction also leads to a marginal decrease in power consumption due to reduced leakage power.

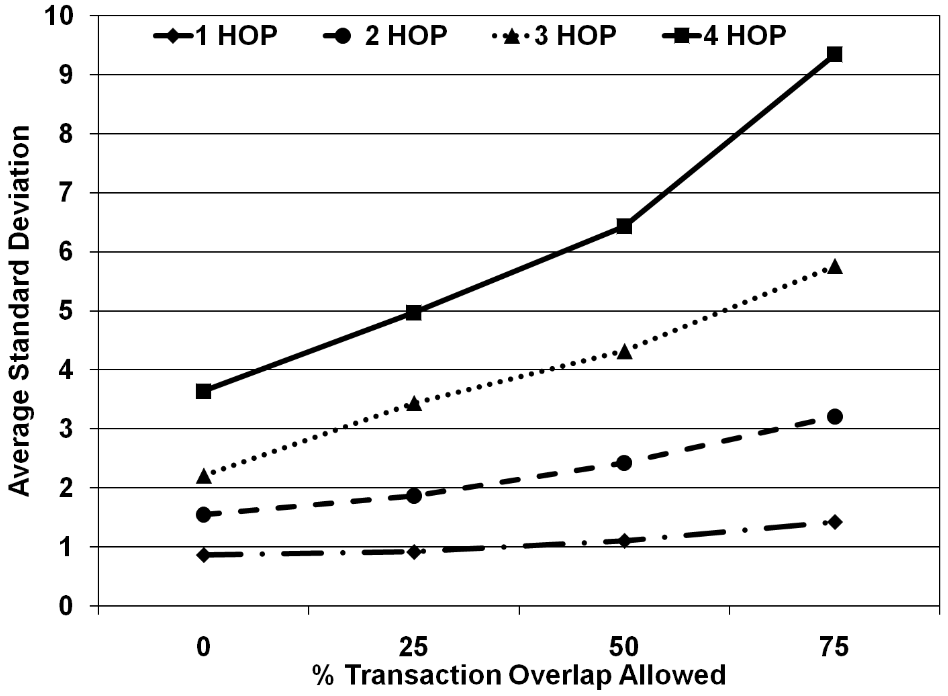


Figure 2.30: Average Standard Deviation with Increasing Transaction Overlap

Figure 2.29 plots the average hop latency for each design. The average per hop latency does not show much increase with 25% overlap. However, the average latency increases sharply for larger percentages of the overlap. The average standard deviation plotted in Figure 2.30 shows a similar trend. Thus, we can consider 25% overlap to be a good trade-off between achievable performance and associated router requirements.

2.5.6 Deadlock avoidance analysis

In this section we present an application example (Figure 2.31) to illustrate the benefit of having deadlock alleviation integrated with synthesis. On the left side of the figure is the floorplan with router allocation and core to router mapping. The layout consists of 9 cores arranged in a 3-by-3 mesh. On the right

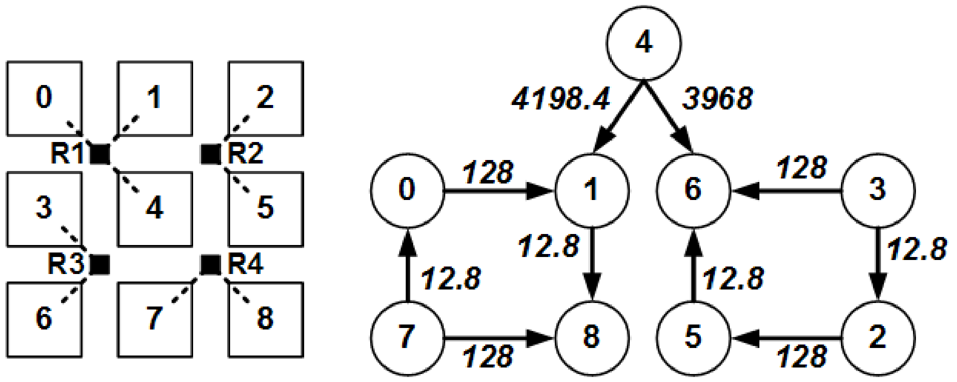


Figure 2.31: Application specification

side of the figure is the communication graph for the application. Each arrow denotes a uni-direction communication trace. Each communication trace is annotated with its bandwidth requirement. There is a high potential for deadlock to exist in the the synthesized NoC due to the cyclical communication pattern $(1- > 8, 5- > 6, 7- > 0, 3- > 2)$. The solution generated by the ILP approach (which was identical to the one generated by GT) is shown in Figure 2.32. The directed edges in the figure denote the routes for the following traces $1- > 8, 5- > 6, 7- > 0$, and $3- > 2$. The potential for deadlock exists in the ILP solution as there is a cycle (due to the cyclical routes of the traces shown in the figure) in the CDG of the NoC. Our approach synthesized the NoC design shown in Figure 2.33 whose CDG does not have any cycles, and therefore is deadlock free.

2.6 Summary

We presented a holistic technique for custom NoC synthesis that can address cumulative bandwidth and transaction level communication requirements, deadlock avoidance, multiple use cases, and router port arity con-

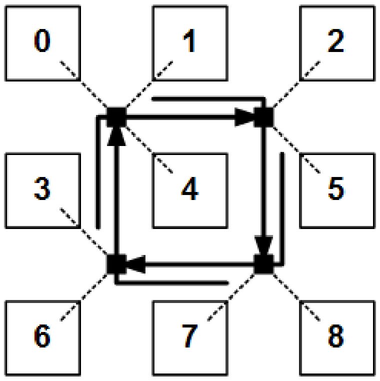


Figure 2.32: ILP solution

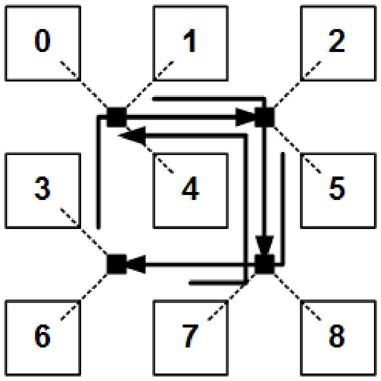


Figure 2.33: Our solution

straints. The solutions generated by the approach use shortest path routes for all communication requirements (minimum dynamic power consumption), and utilize at most twice the number of routers (and leakage power consumption) as the optimal solution.

The experimental results demonstrated that in comparison to ILP and GT techniques our approach is able to generate NoC designs that demonstrate markedly lower average packet latencies and standard deviation with comparable power consumption requirements. The run times of our approach for large benchmarks was 45 mins while ILP and GT required 12 hours (with time out) and 5 mins, respectively. The solutions of our approach demonstrate a reduction in router resource requirements while the power consumption remains comparable as the port arity constraints are increased. As the bandwidth requirements of a design are increased in proportion to the port bandwidth constraint, our approach is able to successfully synthesize NoC designs that show superior performance characteristics, and similar power consumption in comparison to the designs generated by the GT and ILP approaches. We analyzed the impact of degree of overlap permitted between two transactions on the NoC performance, and it was found that 25% overlap gave a good trade-off between the performance of the solution and associated router requirements. Finally, we showed that for input specifications that could result in synthesis of a deadlock susceptible NoC, our approach is successfully able to generate a NoC design which is deadlock free.

Chapter 3

THE SYNTHESIS OF THE MEMORY ARCHITECTURE

Many embedded processor chips aimed at high performance and low power application domains are implemented as multi-processor System-on-Chip (MP-SoC) devices. The multi-media and communication sub-systems of an MP-SoC perform some of the most computation intensive and performance critical tasks, and are key determinants of the system-level performance and power consumption. This chapter presents an automated technique for synthesizing the system-level memory architecture (both code and data) for the streaming sub-systems of an embedded processor. The experimental results evaluate effectiveness of the proposed technique by synthesizing the system-level memory architecture for benchmark stream processing applications and comparisons against an existing approach.

In the next section we motivate the problem. Section 3.1 formally formulates the probelm definition. In Section 3.2 we discuss related work. We present our synthesis technique in Section 3.3. In Section 3.4 present our experimental results and lastly, we conclude the chapter with a summary in Section 3.5.

3.0.1 Motivation

The past decade has seen the emergence of smart mobile devices (smart phones, tablets) as the new technology drivers. Present day versions of these devices support a multitude of applications with voice/data communication, camera, media player, geographical position system (GPS), HD video, and 3D displays on the same device. The processors aimed at such devices must sup-

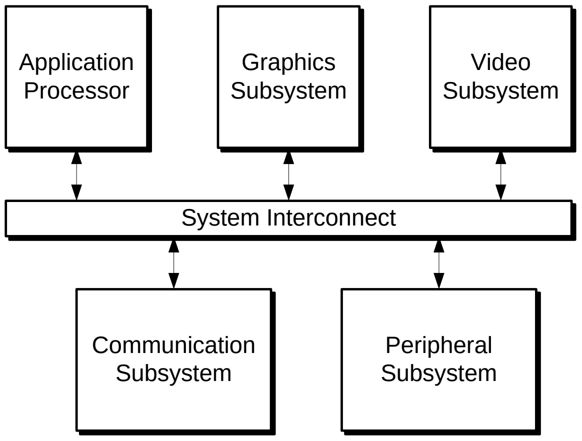


Figure 3.1: Generic MPSoC architecture

port the desired performance while literally "sipping" energy from the battery pack. Further, as smart devices fall in the realm of embedded computing the processors must be designed with a short turn around time. Consequently, chip designers have adopted a heterogeneous System-on-Chip architecture for these processors where each sub-system (application, graphics/media, communication, peripheral) is designed with an optimal constellation of processors, hardware accelerators, memory hierarchy and interconnection network.

Figure 3.1 shows a top-level view of a generic MPSoC aimed at smart mobile devices along with its major sub-systems. The application processor is a general purpose processor such as a dual core ARM Cortex-A9 with cache coherent memory hierarchy. The other sub-systems of the MPSoC are composed of highly optimized instruction processors (such as ARM M3), graphics/DSP processors, and custom hardware accelerators. Further, the non-application sub-systems do not typically support a cache hierarchy and have scratchpad memories for both code and data. The overall architecture is an integration of the various sub-systems via a high bandwidth system-level interconnect.

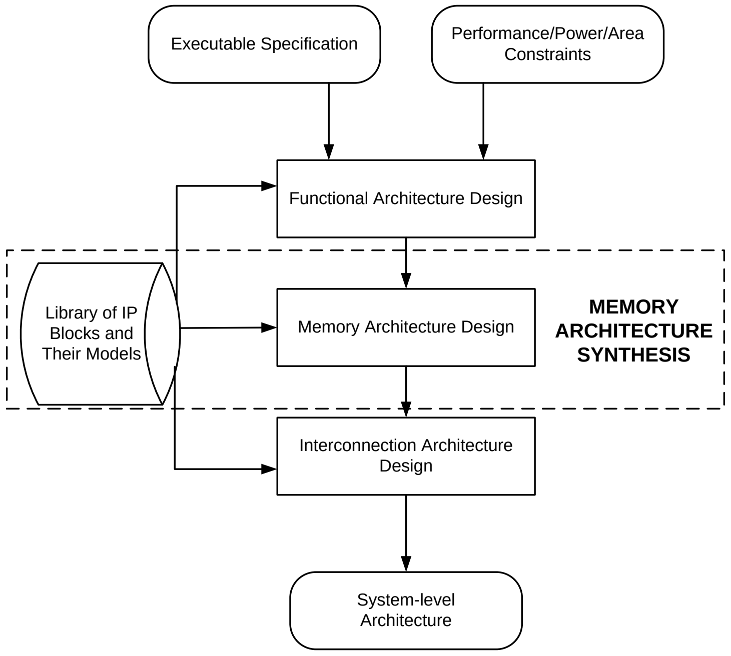

Executable Specification
Performance/Power/Area Constraints
Functional Architecture Design
Library of IP Blocks and Their Models
Memory Architecture Design
MEMORY ARCHITECTURE SYNTHESIS
Interconnection Architecture Design
System-level Architecture

Figure 3.2: System-level design flow

The focus of this chapter is on the system-level architecture design of a sub-system for such a MPSoC. Figure 3.2 depicts the three primary design stages in developing the system-level architecture. The inputs to the system-level design flow are the executable specification, performance/area/power constraints and a library of characterized IP blocks (performance/power/area models). The functional architecture design stage selects the processor core(s) and hardware accelerator(s), and maps the functionality on the processing elements (PE). The memory architecture design stage selects the number and configuration (sizes, ports) of the various memory elements. Finally, the interconnection architecture design stage specifies the topology of the interconnect for the architecture. This work focuses on the design automation of the memory architecture design stage for a domain specific sub-system of a MPSoC.

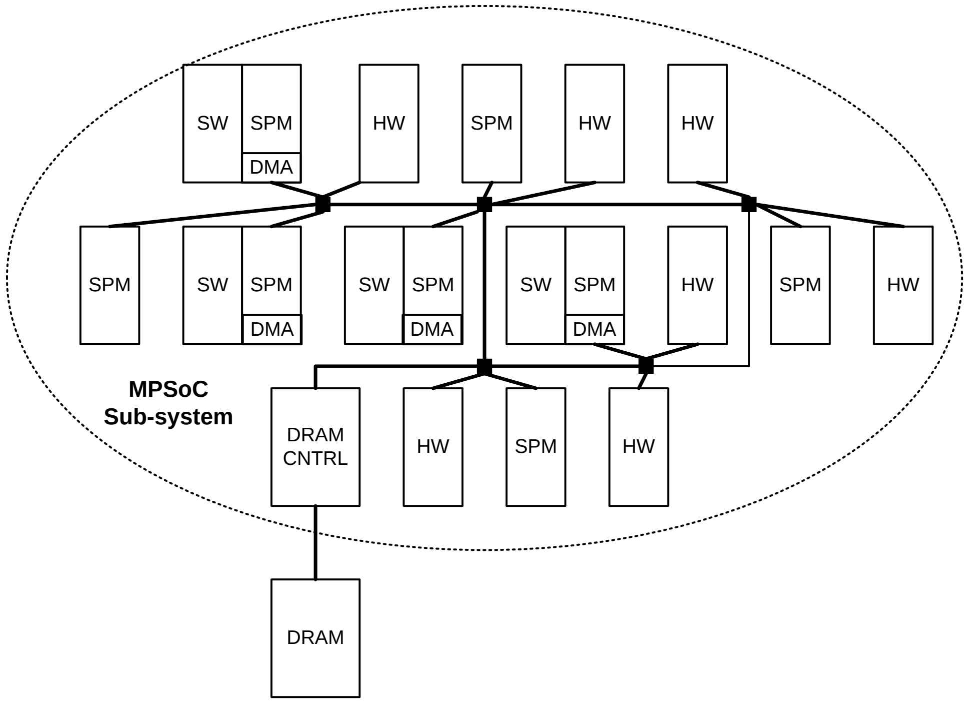


Figure 3.3: Architecture of MPSoC sub-system

The graphics, multimedia and communication sub-systems of the MPSoC depict classical streaming behavior. Consequently, the functionalities of these sub-systems can be most naturally described by stream programming formats. For the purposes of this chapter we assume that the functionality is described by a synchronous dataflow (SDF) specification [11]. As the focus of the chapter is on memory-interconnection architecture synthesis, we assume that the designer performs the functional architecture design stage (selection of PEs and mapping of the SDF actors onto the PEs). Thus, the objective of our synthesis flow is to select the number and configuration (sizes, ports) of the memory elements such that the performance and area constraints are satisfied, and the power consumption is minimized.

Figure 3.3 shows the generic architecture of a sub-system belonging to the MPSoC. In the figure, the instruction processors are denoted by SW, hardware accelerators as HW and the scratch-pad memories as SPM. Each SW PE has a local SPM, and a DMA controller. There may be other SPMs distributed in the architecture that act as shared resources. The various compute nodes and memory elements are connected together by a Network-on-Chip (NoC). The overall performance (and consequently power consumption) of the architecture is a consequence of several design decisions and trade-offs.

As the same SPM is shared by code and data for the SW PE, its performance is dictated by the SDF schedule and code overlay (if required). Code overlay schemes are utilized to minimize the memory required for actor code by mapping the code of multiple actors to the same region of memory. Thus, if the code for the actor to be executed next is not in the memory, it is fetched from DRAM and the currently resident actor code is overlayed. At the system-level the interconnect delays are dictated by the topology and the DMA schedules. As the objective is to minimize the power consumption subject to both performance and area constraints the number (and sizes) of memory elements, and router nodes that can be utilized are limited. In our approach we utilize an existing NoC synthesis technique [12]. This chapter presents a novel automated memory synthesis approach that is able to effectively perform all the various trade-offs, and consequently generate a highly optimized memory and NoC architecture for the sub-system.

3.1 Problem Definition

The formal definition of the problem is as follows. Given:

1. a synchronous dataflow specification of a streaming application: A Directed Graph $G(V, E)$, where $v \in V$ is a set of filters or actors, and the set of directed edges $e(u, v) \in E$ denotes that the data produced by u is consumed by v. Each directed edge $e(u, v) \in E$ is annotated with the size of the data block, $\delta(u)$, produced by filter u and the size of the data block, $\phi(v)$, consumed by filter v.

2. a set PEs and a mapping of filters to the PEs: A bipartite Graph $G(V, P, M)$, where $v \in V$ is the set of filters pertaining to the streaming application, $p \in P$ is the set of PEs (HW or SW), and the set of undirected edges $e(v, p) \in M$ denotes the mapping of filter v onto PE p. In the case of SW PEs more than one filter may be mapped to it. Each filter $v \in V$ is annotated with the code size of the filter, $\omega(v)$, and the execution time of the filter, $\tau(v)$.

3. performance and area constraints: Designer specified throughput constraint on the SDF, and area constraint on the sub-system.

4. library of characterized memory elements: A library consisting of memory elements parameterized in terms of size and number of ports, and characterized in terms of power consumption, area requirement, and access latencies.

5. library of characterized NoC router architectures: A library of NoC IP components (routers and network interfaces) characterized in terms of power consumption, area requirement, and no load latency.

Synthesize

1. a memory architecture for the sub-system: The memory architecture specifies the number and configuration of distinct SPM elements in the sub-system.

2. a NoC topology: The NoC topology specifies the number and configuration of the routers used in the architecture, and their interconnection to the PEs and memory elements.

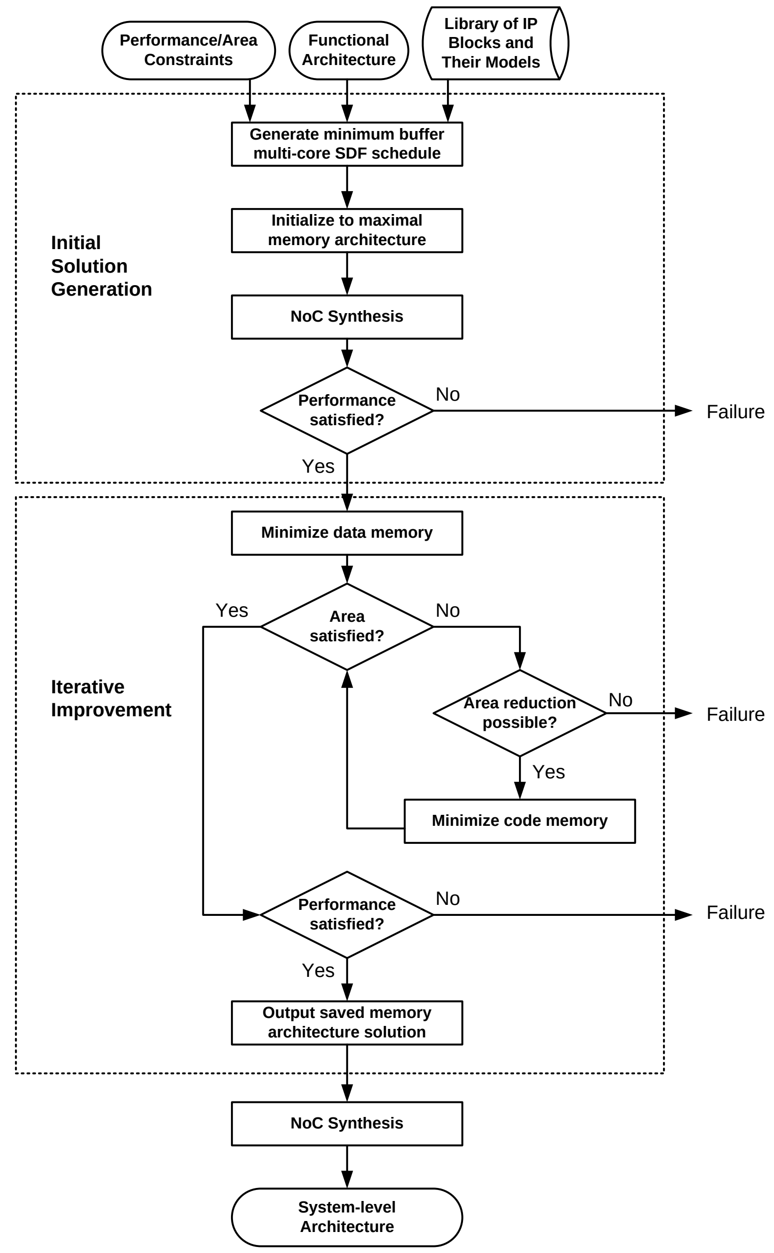
3. a memory usage description for the sub-system: The memory usage description describes the utilization of the various SPMs for actor code and data blocks. The description specifies if a code overlay scheme has been utilized for the SPM, and if indeed it has been utilized, the description includes a mapping of actors to region and segments in the SPM. The usage description also defines a mapping of the actor data blocks to memory regions of various SPMs. Further, as the memory usage is minimized by utilizing shared SPM for ephemeral data, more than one data block may be assigned to same region of a SPM.

4. a execution schedule for SDF and DMA: The execution schedule gives the global schedule for firing of various actors, and launching of DMA operations for code overlays, and data transfers.

such that the performance and area constraints are satisfied, and the power consumption of the design is minimized.

3.2 Related Work

System-level MPSoC memory synthesis has been attracting growing attention over the past few years. A representative selection of existing work is discussed in this section. Meftali et al. [13] presented an integer linear programming approach for memory synthesis that focused only on data blocks. Pasricha et al. [14] proposed an integrated heuristic approach for memory and bus matrix synthesis that was also primarily aimed at data blocks. Pandey et al. [18] presented a bus and data memory architecture co-synthesis approach based on slack allocation. Issenin et al. [15] proposed a MILP and heuristic memory synthesis approaches that utilized a fixed topology bus architecture and aimed at minimizing data memory usage. An extension of the same work for mesh based NoC was also proposed [16]. Monchiero et al. [17] presented the results for design space exploration of a non-uniform memory access architecture interconnected with a parameterized (ring, spidergon or mesh) NoC fabric. Recently, Lee et al. [19] presented an approach for integrated MPSoC synthesis for SDF specification that considered pre-selected bus templates. In contrast to these approaches we consider NoC aware memory architecture design for streaming applications. Further, we not only optimize and account for data block memory usage but also consider the impact of code memory optimization. Specifically, we consider the design trade-offs for partitioning the same SPM between code and data. We also consider the performance and power overheads of code overlay schemes that can reduce the memory requirements (and consequently the MPSoC area). To the best of our knowledge the system-level memory synthesis approach presented in this chapter is the only technique that considers the impact of both data and code memory requirements during design space exploration.



Performance/Area Constraints
Functional Architecture
Library of IP Blocks and Their Models
Initial Solution Generation
Generate minimum buffer multi-core SDF schedule
Initialize to maximal memory architecture
NoC Synthesis
Performance satisfied?
No
Failure
Yes
Iterative Improvement
Minimize data memory
Area satisfied?
Yes
No
Area reduction possible?
No
Failure
Yes
Minimize code memory
Performance satisfied?
No
Failure
Yes
Output saved memory architecture solution
NoC Synthesis
System-level Architecture

Figure 3.4: Top-level view of memory synthesis

3.3 System-level memory synthesis

The top-level view of our memory synthesis technique is shown in Figure 3.4. The overall strategy of our technique is to begin with a highest performance and lowest power consuming solution. We then iteratively arrive at a solution that satisfies an area constraint with minimal decrease in performance and increase in power consumption. The inputs to the memory synthesis design stage are the i) performance and area constraints, ii) functional architecture description, and iii) the library of memory and interconnect IP blocks along with their power, performance, and area models. The memory synthesis technique broadly consists of two stages, an initial solution generation step followed by an iterative improvement stage.

Initial solution generation stage: As a first step we generate a minimum buffer usage multi-core SDF schedule. We utilize a well known heuristic approach to generate the schedule [20]. We then consider a maximal memory architecture for the sub-system. In the maximal memory architecture the local SPM of each SW PE is large enough to host the entire code base of all the actors assigned to the PE. Further, there is sufficient memory for double buffering of inter-PE transfers. Finally, we do not perform any memory optimization for ephemeral data blocks. Thus, the maximal memory architecture represents the maximum SPM memory that is required for the design. Consequently, the design also depicts the best possible performance and minimal power consumption[1]. We then synthesize the NoC architecture for the sub-system. As mentioned earlier we utilize an existing approach to synthesize the NoC [12]. The NoC synthesis technique supports guaranteed throughput traffic which is

[1]Power consumption is minimal because the number of accesses to DRAM is minimal.

ideal for streaming applications. The synthesis technique includes a system-level floorplanning stage, and is thus able to generate very good estimates for communication latencies and power consumption. The NoC synthesis technique minimizes both the power consumption (primary goal) and resource requirement (secondary goal) of the interconnection architecture subject to the communication bandwidth requirements. Finally, we evaluate the performance of the design and verify if the performance constraint is satisfied. As the initial design represents the best performance design, we declare failure if the performance constraint is not satisfied. Alternatively, if the performance constraint is satisfied we enter the iterative improvement stage in which we aim to satisfy the area constraints, and minimize power consumption.

Iterative improvement stage: The objective of the iterative improvement stage is to satisfy the area constraints and minimize power consumption. As a first step we minimize the memory required for ephemeral data by analyzing their lifetimes and mapping them to the same memory region wherever possible. We introduce shared SPM into the memory architecture if the data blocks that share the memory region are from different PEs. Notice, that data memory reduction does not have an appreciable impact on the performance[2]. However, the power consumption is expected to increase due to an increase in NoC communication. We next check if the area constraint is satisfied. If it is we have successfully synthesized the memory architecture. Alternatively, we try to further reduce the memory requirement by introducing code overlays. Introduction of code overlay involves periodic fetching of code from the off-chip DRAM memory, and it slightly increases the power consumption and

[2]Mapping the data blocks to remote SPM may introduce additional communication delays. However, the NoC synthesis technique is able to generate designs with minimum latency, and consequently the performance impact is minimal.

```
$minimize\_data\_memory()$
{
$G = generate\_interference\_graph()$
$clique\_partitioning(G)$
for each clique $C$ in $G$ do
    $combine\_data\_blocks(C)$
end for
}
```

Figure 3.5: Data memory minimization pseudo-code

reduces the performance. The impact of performance reduction can be amortized to some extent by scheduling code pre-fetch DMAs whenever possible. Code overlay is only introduced if the area constraints are not satisfied. We iteratively reduce the code memory usage (increase code overlay overheads) until either the area constraints are satisfied or no further reduction in memory can be achieved. In the case of the later we declare failure as the area constraints are not satisfied. If they are satisfied we again evaluate the performance constraint. If the performance constraint is still satisfied we declare success and output the memory architecture. Alternatively, we declare failure due to non-satisfaction of the performance constraint. In the following two sub-sections we discuss the data and code memory minimization stages in further detail.

3.3.1 Data memory minimization

The objective of the data memory minimization stage is reduce the memory requirement for ephemeral data blocks by analyzing their lifetimes, and assigning them to the same memory region. We utilize a classical clique partitioning algorithm to achieve our goal. The pseudo-code for data memory minimiza-

```
clique_partitioning(G)
{
while vertex exists with degree greater than zero do
    $V = smallest\_non\_zero\_degree\_vertex()$
    $U = smallest\_degree\_attached(V)$
    $N = combine\_vertices(U, V)$
    for each vertex P attached to V do
        for each vertex L attached to U do
            if P equals L then
                add edge from N to P
            end if
        end for
    end for
    $update\_degrees()$
end while
return partitioning
}
```

Figure 3.6: Clique partitioning pseudo-code

tion stage is shown in Figure 3.5. We first generate an interference graph (Line 3, Figure 3.5). The interference graph is specified as $G(V, E)$ where V is the set of data_blocks and E is the set of edges from (u, v) where u and v are vertices in V. An edge (u, v) exists when there is no interference between data blocks u and v. Interference is defined as both data blocks being alive during a portion of the same time frame. A data block is alive from the time when it first begins to be written to, and up to (and including) the last time instance that it is read from. As the data blocks may be present in distinct SPMs, we annotate each edge $(u, v) \in E$ with the physical distance between the two distinct SPMs, d. Notice that we do synthesize a NoC as part of the initial solution, and our NoC synthesis technique generates a floorplan as part of its design flow. Consequently, we can deduce the distance between two distinct

SPMs. The distance is used as a tie breaker during the clique partitioning stage.

As a next step (Line 4, Figure 3.5) we invoke the clique partitioning algorithm (Figure 3.6). The algorithm begins by finding the vertex with the smallest non-zero degree (Line 4, Figure 3.6). The degree of a vertex is equal to the number of edges incident on the vertex. The algorithm then finds the smallest degree vertex that is attached to the previously found vertex (Line 5, Figure 3.6). If there is a tie between vertices the algorithm will choose the vertex with the highest common neighbors as the first vertex. If there is still a tie the algorithm will choose the vertex that has the smallest physical distance d (remember this is annotated on the edge). The algorithm will then combine these two vertices into a single vertex. Next the algorithm updates the edges of the graph. An edge will exist from the new compound vertex to another vertex if and only if the vertex was connected to both the vertices that have been collapsed into the compound node (Line 10, Figure 3.6). The degrees of the vertices are updated and the algorithm repeats until all vertices have a degree of zero.

The $minimize\_data\_memory()$ algorithm then collapses the data blocks which are part of a clique into a single SPM (Line 4, Figure 3.5). Notice that at this stage we might introduce new shared SPMs if the data blocks were originally resident on local SPMs of distinct PEs.

3.3.2 Code memory minimization

We invoke the code memory minimization stage only if the area constraints are not satisfied. The objective of the code memory minimization stage is reduce

```
1: $minimize\_code\_memory()$
2: {
3: Initialize each filter to occupy its own region
4: $calculate\_IF()$
5: while area constraint not met and $\mid R \mid$ greater than 1 per SPM do
6:    $collapse\_smallest\_IF()$
7:    $update\_IF()$
8: end while
9: $perform\_segmentation()$
10: }
```

Figure 3.7: Code memory minimization pseudo-code

the code memory requirements for SW PEs by off loading code to DRAM. We would like to emphasize that the code is always resident in the DRAM. In the initial solution generated by our approach the entire code base is fetched in to the on-chip SPM before the start of the first iteration. Consequently, in the initial solution for we do not need to fetch code from DRAM for any subsequent iteration of the SDF. In the code memory minimization stage we assign code bases of two or more filters to the same region of the memory. Thus, during an iteration of SDF execution, we would have to fetch code for one or more filters from the DRAM. Therefore, there is both a performance and power (as accessing DRAM consumes a lot more power) penalty associated with code memory reduction.

The pseudo-code for the code minimization algorithm is given in Figure 3.7. The algorithm begins by initializing each filter to its own unique region (Line 3). We next calculate the interaction factor (IF) for each region pair (Line 4). The IF is first initialized to zero for all region pairs. Next we step through the SDF execution schedule, and for each switch from region r_i to

Table 3.1: Benchmark Specifications

Benchmarks	#Actors	#Edges	#Executions
Beamformer	40	72	64
Bitonicsort	26	31	68
DCT	15	22	28
FFT	17	16	58
Filterbank	51	65	94
Fmradio	29	39	58
Average	30	41	62

region r_j or vice versa the IF(r_i, r_j) is increased by one. The IF for regions on distinct SPMs is initialized to infinity. Next, the algorithm enters a loop if the area constraint has not been met, and there is at least one SPM with 2 or more regions. Within the loop the algorithm collapses the region pair with the smallest IF. The IF of the regions is then updated and the loop repeats. Upon exiting the loop, the algorithm performs segmentation on the regions where two or more filter belonging to a single region are assigned to the same segment. Segmentation amortizes the DMA cost for fetching the code bases of the filters from the DRAM.

3.3.3 Time Complexity Analysis

The time complexity of finding the minimum buffer schedule is $O(n)$, where n is the number of actors. The time complexity of the minimizing the data is $O(b^3)$, where b is the number of data blocks. And lastly, the time complexity of minimizing the code is $O(n^4)$, where n is the number of actors. Therefore, the total time complexity for the memory architecture synthesis is $O(n + b^3 + n^4)$. Typically, the number of data blocks is substantially larger than the number of actors and therefore in practice the time is dominated $O(b^3)$.

3.4 Experimental Results

We evaluated the efficacy of our proposed memory synthesis approach by considering the design of sub-systems that implemented six benchmarks from the StreamIt [21] suite. The benchmarks are described in Table 3.1. In the table the second and third columns denote the number of actors and edges in each benchmark, and the last column denotes the total number of actor firings in one iteration of the SDF. We generated MPSoC designs for each benchmark by considering 4, 6, 8, 12, and 16 PEs. For each number of PEs we set the throughput constraint to be 0.75 times the throughput of the initial baseline solution. We then iteratively reduced the area constraint until we had the tightest area constraint for each benchmark in which our technique was able to generate a valid design. We compared the solutions generated by our technique with the initial baseline solutions as well as with the designs generated by the existing 2-stage technique proposed in [16]. Our technique took on average 15 minutes to generate the designs which is reasonable considering we perform NoC synthesis which contains a floorplanning stage.

3.4.1 Comparison against Baseline Solution

The first set of experiments we compared the designs generated by our technique after the final NoC synthesis stage with the baseline initial solution for each benchmark. Figures 3.8, 3.9, and 3.10 plot the normalized area, throughput, and performance per watt of the various designs. For each benchmark in the plot, the results are normalized to the initial baseline (or maximal area) solutions of the 4 PE design. For example, the area plots for the beamformer benchmark designs are normalized to the area of the initial baseline solution for the beamformer implemented with 4 PEs. For some benchmarks (dct and

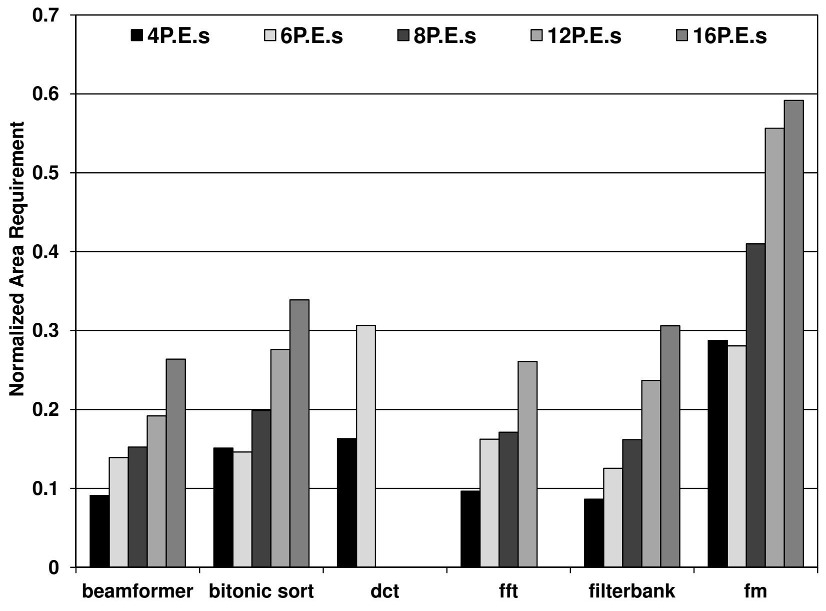


Figure 3.8: Normalized area

fft) we do not plot results for all PEs as the benchmarks were too small to be mapped onto the larger number of PEs.

In Figure 3.8, we see that our technique is able to generate designs with very tight area constraints. With the smallest area constraint at 4 PEs being 10% for the 'beamformer' benchmark and the largest constraint being 30% for 'fmradio' benchmark. On average, across all benchmarks our technique is able to generate designs that require 75.3% less area than the initial baseline solutions for a 25% loss in performance. We also see that the area requirement compared to the initial 4 PE design increases as we increase the number of cores. This is due to the increase in the required amount of SPM memory for the cores (each core requires a minimal amount). In Figure 3.9, we notice

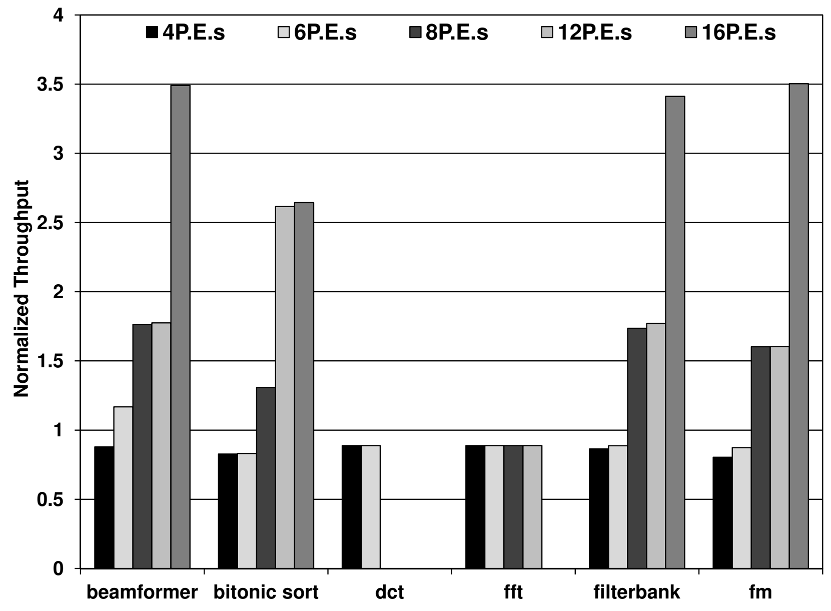


Figure 3.9: Normalized throughput

that for the initial 4 PE design the throughput of the designs generated by our technique is slightly lower than the initial solution. This is to be expected due to the code overlay overhead from DRAM accesses to retrieve code. However, we see as the number of PEs increases we gain a substantial increase in throughput over the initial baseline solution. Figure 3.10 illustrates that the designs generated by our technique have higher performance per watt than the initial baseline solutions. At 16 PEs the performance per watt of our design is almost 2 times the intitial 4 PE baselin solution for both 'bitonic sort' and 'fm.' And in the other three benchmarks at 16 PEs, our designs had higher performance per watt than the initial baseline 4 PE solutions.

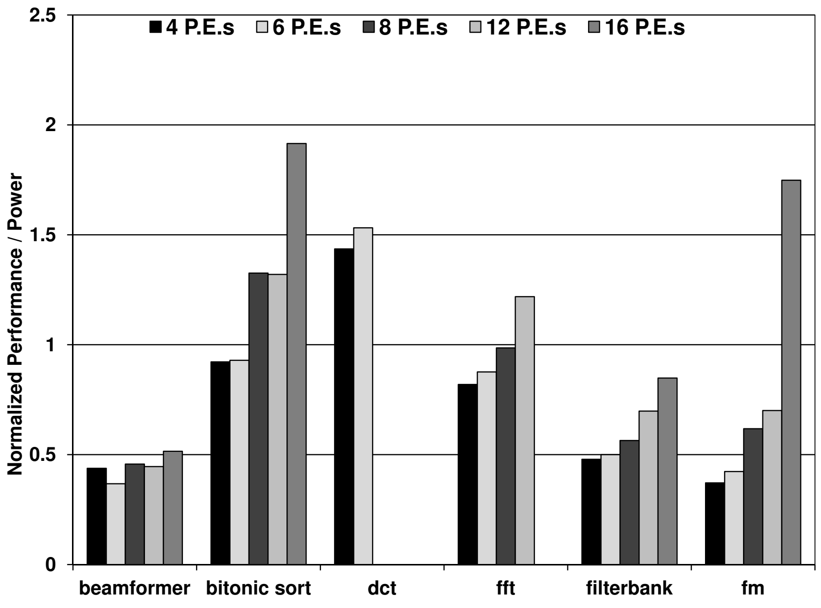


Figure 3.10: Normalized Perf./Watt

3.4.2 Impact of Code Overlay Optimization

Figure 3.11 demonstrates the impact of the code overlay optimization for two benchmarks with the maximum number of actors (namely beamformer and filterbank). The plot depicts normalized throughput, energy and area for 16 PE designs. The plots are normalized to the solutions that only apply data optimizations and do not apply code overlay. As is depicted in the plot, the code overlay optimization is able to considerably reduce the area requirements (by over 50%) for comparable performance. The trade-off is the increase in energy due to code overlay accesses to DRAM. Particularly for the filterbank application the increase in energy is only about 30%. Area minimization is critical

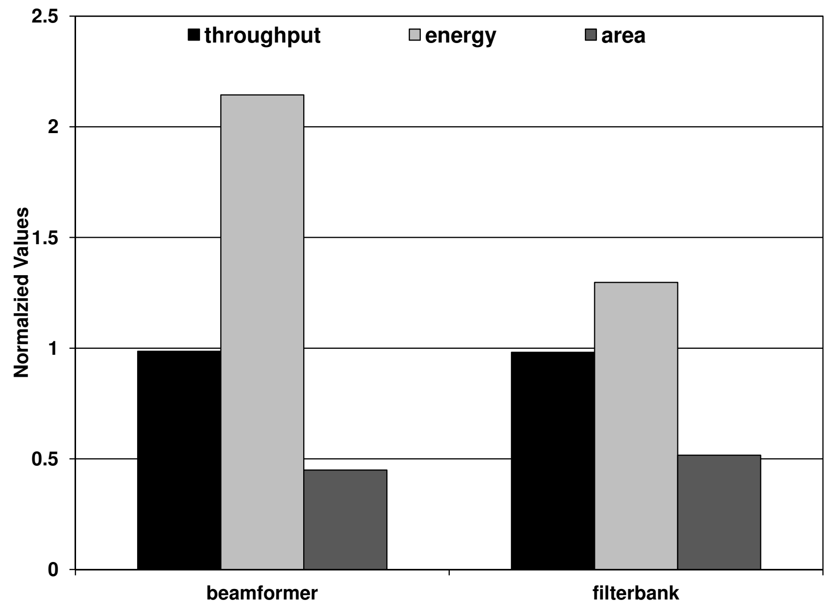


Figure 3.11: Impact of code overlay

as the silicon real estate determines the cost of manufacturing. Code overlay optimization is able to generate design alternatives for tight area constraints that would not be otherwise possible.

3.4.3 Comparison with Existing Approach

Figure 3.12 compares the designs generated by our technique against a 2-stage synthesis technique presented in [16]. The technique proposed in [16] only accounts for data memory optimization (at the fine grain). Also, the technique considers a mesh (template) topology for the NoC network. The technique generates a data reuse graph consisting of data buffers in a hierarchical manner with each higher level buffer containing all of the data in the buffers below it in the hierarchy. The technique then greedily selects buffers to add to the design

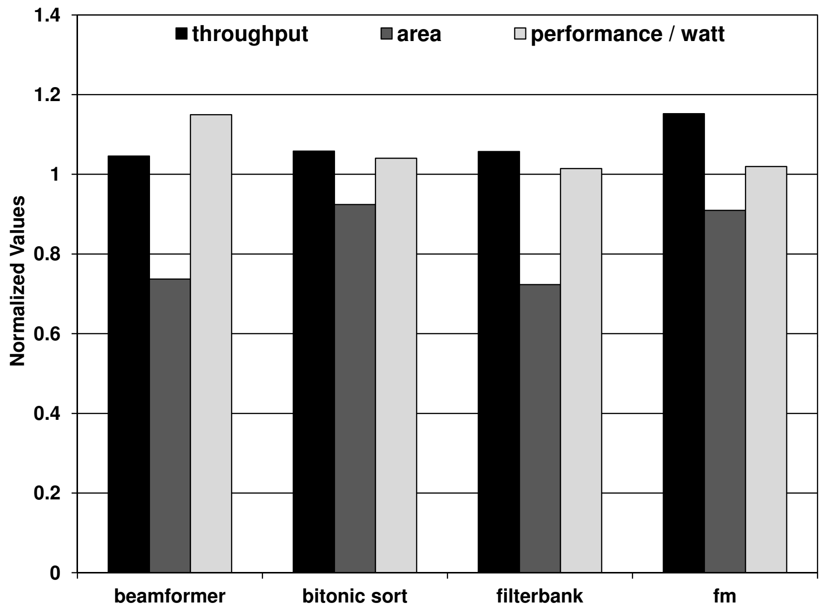


Figure 3.12: Existing approach

based on the energy savings of using the buffer. We modified the technique to use the larger data blocks present in SDF specifications. We also modified the technique to use the same NoC synthesis tool that we use. This will ensure a fair comparison between the techniques.

Figure 3.12 plots the normalized throughput, area, and performance per watt for 4 StreamIt benchmarks. The plots are normalized to the respective values for the designs generated by the existing approach [16]. As can be observed in the figure, our technique consistently gives better performaning designs that utilize lower area and have higher performance per watt. On an average our designs show 7.8% increase in performance, 17.7% reduction in area and 5.6% increase in the performance per watt. Our technique is able to

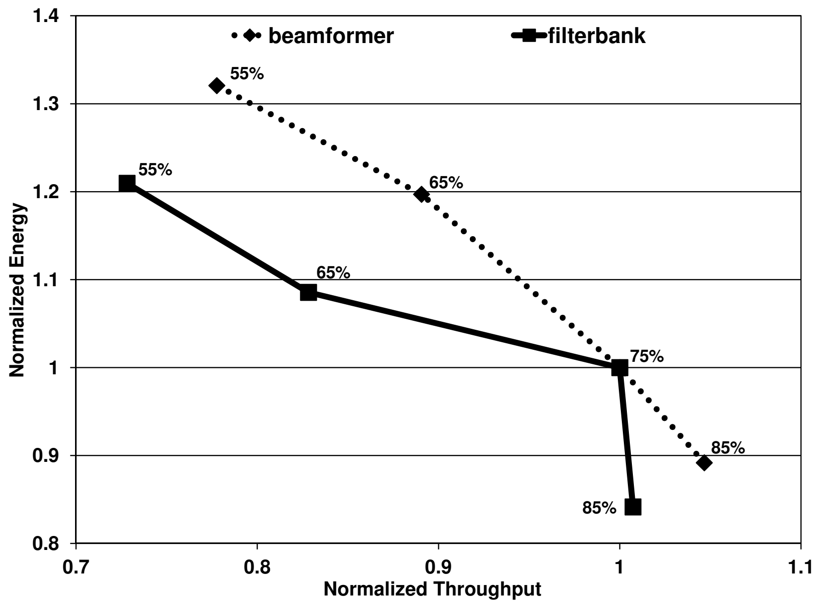


Figure 3.13: Area impact

give better results because of more comprehensive data minimization methods and incorporation of code overlay optimizations.

3.4.4 Impact of Area Constraint

In our last experiment, we evaluated our approach by varying the area constraints for the 12 PE designs. In this experiment we only considered 2 benchmarks, and Figure 3.13 plots the results. In the plot each point (energy and throughput) depicts the design obtained for the respective area constraints (55%, 65%, 75%, 85%). The area constraint is achieved by percentage scaling the area for the initial baseline (maximal area) solution. The plots are normalized to the 75% area constraint design. From the plot, we can see that as the area constraint is made tighter the throughput of the designs decreases and the energy consumption increases. This is expected due to the increase in code overlay overheads as more code is forced into main memory.

3.5 Summary

We presented an approach for synthesizing the system-level memory of a MP-SoC sub-system that demonstrates streaming characteristics. The approach accounts and optimizes for the memory requirements for both code and data. We evaluated our approach by extensive experimentation with streaming application benchmarks through comparisons with an existing approach and the initial baseline solution. Our technique performed superiorly to the existing approach and clearly demonstrated the ability to generate high quality designs meeting the area and performance constraints while maintaining a low energy consumption.

Chapter 4

THE SYNTHESIS OF THE FUNCTIONAL ARCHITECTURE

Recently multi-processor System-on-Chip (MPSoC) has e-merged as the architecture of choice for high performance, low power embedded devices. The sub-systems of an MPSoC perform highly computation intensive and performance critical tasks. These sub-systems are key determinants of the system-level performance and power consumption. This chapter presents an automated technique targeted at the synthesis of the system-level functional architecture for streaming sub-systems of an embedded processor. Specifically, the selection of processing elements in the sub-system and the mapping of the application tasks onto the processing elements. The experimental results evaluate the effectiveness of the proposed technique by synthesizing HW-SW system-level functional architectures for streaming benchmarks and through comparisons against both pure software and pure hardware designs.

In the next section we motivate the problem. In Section 4.2 we formally define the problem. We discuss related work in Section 4.3. In Section 4.4 we discuss the synthesis technique in detail. In Section 4.5 we present our experimental results. Lastly, we summaize the chapter in Section 4.7.

4.1 Motivation

Recently the demand for high performance, power efficient embedded systems (cell phones, set-top boxes, etc.) has grown substantially. As the demand for higher performance embedded systems increases Multi-Processor System-on-Chips (MPSoCs) are becoming a popular solution to address these demands.

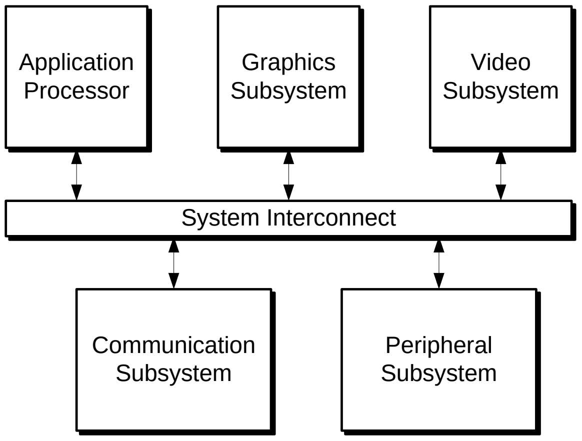


Figure 4.1: Generic MPSoC architecture

Most real-world MPSoCs consist of a compilation of heterogeneous processing elements (PEs) (general purpose processors and application-specific integrated circuits (ASIC)) on a single die [26] [27]. This architecture is appealing to designers due to its native ability to provide significant parallelism [25] to meet the demands of the application.

Figure 4.1 depicts a generic top-level view of a MPSoC. The application processor is a general purpose processor such as an ARM Cortex-R4 with cache coherent memory hierarchy. The other sub-systems of the MPSoC are composed of highly optimized instruction processors, graphics/DSP processors, and custom hardware accelerators. Typically, the non-application subsystems do not support cache coherency and instead have scratchpad memories for both code and data. The overall architecture is an integration of the various sub-systems via a high bandwidth system-level interconnect.

Figure 4.2 illustrates the three main stages in the MPSoC design process. The inputs to the design flow are the executable specification, performance, area, and power constraints and a library of characterized IP blocks

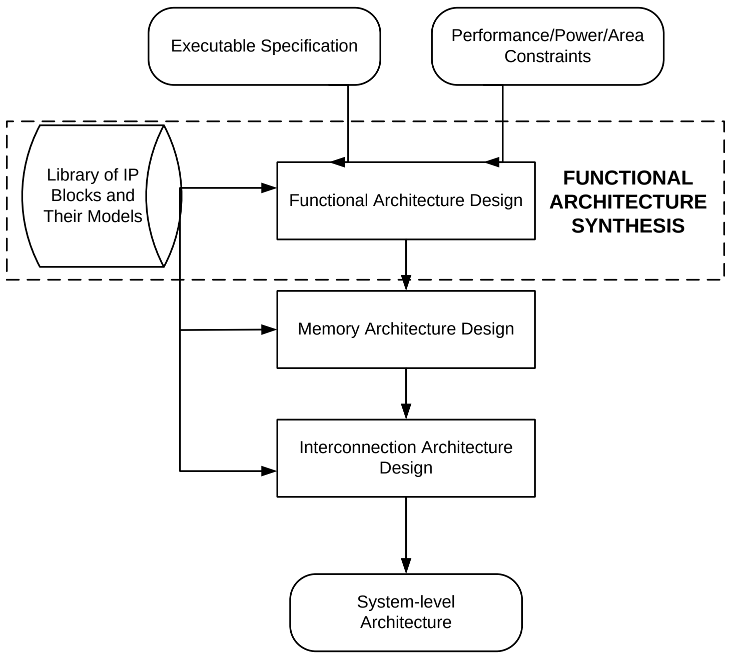

Executable Specification
Performance/Power/Area Constraints
Library of IP Blocks and Their Models
Functional Architecture Design
FUNCTIONAL ARCHITECTURE SYNTHESIS
Memory Architecture Design
Interconnection Architecture Design
System-level Architecture

Figure 4.2: System-level design flow

(performance/power/area models). The functional architecture design stage selects the processor core(s) and hardware accelerator(s), and maps the functionality of the application onto the processing elements (PE). The memory architecture design stage selects the number and configuration (sizes, ports) of the various memory elements. Finally, the interconnection architecture design stage specifies the topology of the interconnect for the architecture. The work in this chapter focuses on the design automation of the functional architecture design stage.

During the functional architecture design stage the selection of the hardware and software processing elements is performed and the application is mapped onto these processing elements. Figure 4.3 illustrates a generic design of a sub-system. The hardware accelerators are denoted by HW, the software processing elements are denoted with a SW, and the scratchpad

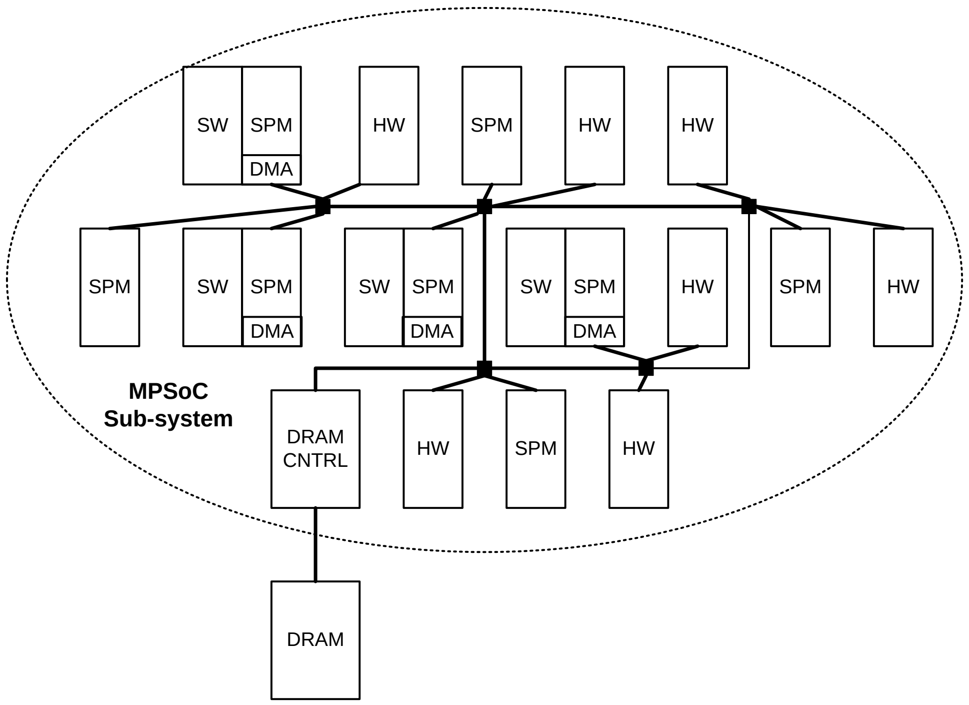


Figure 4.3: Architecture of MPSoC sub-system

memories are denoted by a SPM. The processing elements communicate via an interconnect.

The selection of the hardware and software processing elements impacts the performance, area, and flexibility of the final design. While a purely hardware design exhibits high performance, low area, and low power it fails to provide any flexibility after the design reaches the market. A pure hardware design will only be able to perform the task it was originally designed for. Further, a pure hardware design typically requires a longer design time in order to ensure the hardware accelerators function properly. While a pure software design typically requires a high amount of area and power while giving varying performance. However, a pure software design has a short time to market and provides extensive flexibility. Designers can simply change the software being executed in order to add or remove functionality. Typically, a design with priority given to software cores is desired. This design will provide the performance required through the use of hardware accelerators, with moderate area and power requirements while still allowing for a short time to market and future flexibility. In this chapter, we present a novel functional architecture synthesis technique that is capable of exploring designs ranging from pure software to pure hardware and consequently generate highly optimized functional architectures for MPSoC sub-systems. Additionally, our technique provides a memory sub-system for the functional architecture.

4.2 Problem Definition

The formal definition of the problem is as follows. Given:

1. a synchronous dataflow specification of a streaming application: A Directed Graph $G(V, E)$, where $v \in V$ is a set of filters or actors, and the set of directed edges $e(u, v) \in E$ denotes that the data produced by u is consumed by v. Each directed edge $e(u, v) \in E$ is annotated with the size of the data block, $\delta(u)$, produced by filter u and the size of the data block, $\phi(v)$, consumed by filter v. Each filter $v \in V$ is annotated with the code size of the filter, $\omega(v)$, if the filter is placed in software.

2. performance and area constraints: Designer specified throughput constraint on the SDF, and area constraint on the functional architecture.

3. library of characterized processing elements: A library consisting of processing elements categorized into software PEs and ASIC PEs. The ASIC PEs, which perform the function of one filter, are parameterized with

the filter function it performs, the execution time, energy consumption, and the area requirement. The software PEs are parameterized with frequency, area, and energy requirement.

4. library of characterized memory elements: A library consisting of memory elements parameterized in terms of size and number of ports, and characterized in terms of power consumption, area requirement, and access latencies.

synthesize

1. a functional architecture for the system: The functional architecture specifies the number and type of processing elements in the sub-system.
2. a mapping of the SDF to the processing elements: A mapping of each filter in the SDF to the processing elements in the sub-system.
3. a memory architecture for the sub-system: The memory architecture specifies the number and configuration of the SPM elements in the sub-system.

such that the performance and area constraints are satisfied, software cores are given priority, and the power consumption of the design is minimized.

4.3 Related Work

The work presented in this chapter for the synthesis of the functional architecture looks at the multiprocessor scheduling problem. The multiprocessor scheduling problem has been researched quite extensively: [31] [32] [33] [37]

[35] [36] [28] [29] [30]. However, most of these approaches assume a fixed number of processors and schedule the tasks onto the processors. In Fernandez et al. [33], a upper and lower bound on the number of processors is presented such that the time of the critical path is not exceeded. In Kasahara et al. [36], heuristic algorithms are presented to minimize the execution time. However neither of these, [33] and [36], consider the communication overhead or the memory requirements of code and data. In our technique, we consider the communication overhead of the interconnection and memory architectures. Additionally, we consider the tradeoffs of placing filters (tasks) in hardware or software.

Several works have been proposed to address the hardware-software synthesis problem of the functional architecture. Optimal synthesis of the functional architecture with hardware and software PEs is a NP-complete problem [37] and therefore techniques that use integer linear programming [38] [39] or use exhaustive design space exploration [40] can only be applied to very small design instances.

Since, optimality is too difficult to achieve several heuristics have been presented. In Dick et al. [41], a genetic algorithm is presented to solve the hardware-software synthesis problem. While this approach handles the selection of PEs as well as the mapping of the application task graph to the PEs, the approach does not consider any area constraints or memory requirements. In Chen et al. [42], a SA-based algorithm is presented to perform the selection of the PEs and the mapping of the task graph. However, again this approach does not consider the memory requirement of the PEs and consequently the area requirement. In Chen et al. [43], a heuristic is presented to perform the co-synthesis of the PEs and memory sub-system. The approach addresses the mapping of the task graph on to the processing elements. However, the approach limits the design space to a designated NxN mesh NoC architecture where either a PE or a memory is attached to each NoC router.

In the work in this chapter, we present a heuristic capable of synthesizing a heterogenous sub-system consisting of both hardware and software PEs and will account for the memory requirements of the PEs. Additionally, the heuristic will account for area and performance constraints while minimizing power consumption. Further, the technique will be integrated with the memory architecture design stage to provide a co-synthesis design flow.

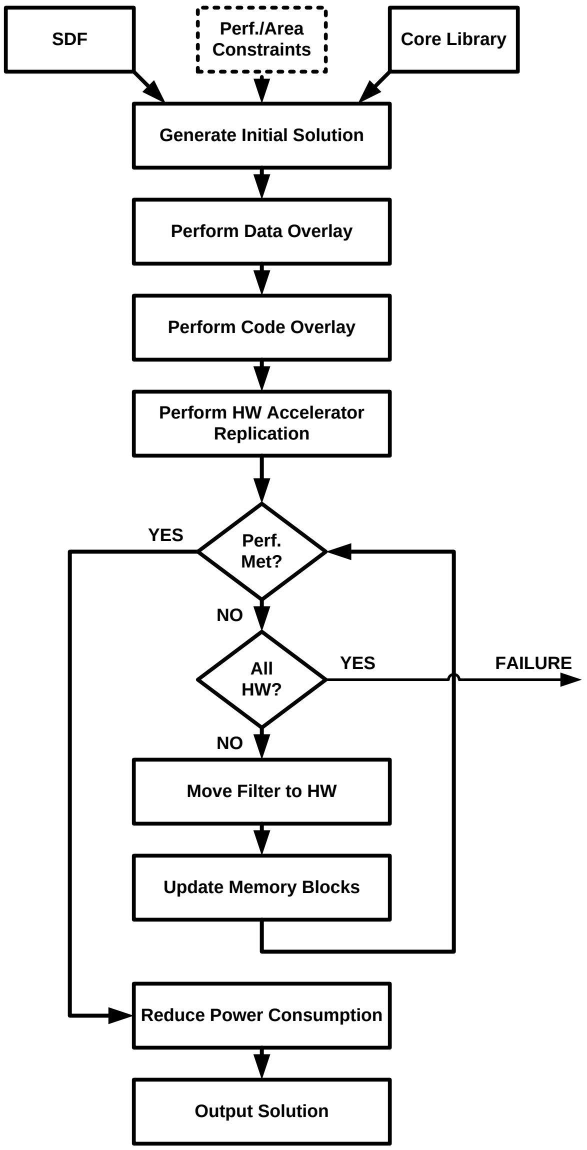


Figure 4.4: Functional Architecture Synthesis Flowchart

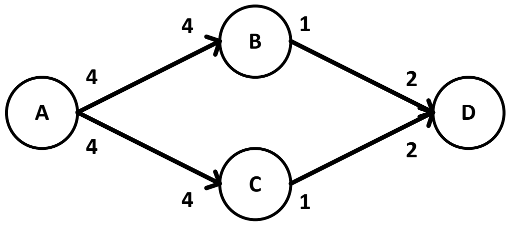


Figure 4.5: Example SDF Specification

4.4 Functional Architecture Synthesis

The top-level view of our functional architecture synthesis technique is shown in Figure 4.4. The overall strategy of our technique is to begin with a pure software solution that meets the area constraint. We then iteratively arrive at a HW-SW solution that meets both the area and performance constraints. We then reduce the power consumption if possible. The inputs to the functional architecture synthesis technique are the i) SDF specification of the streaming application, ii) a library of the processing elements and memory blocks along with their power, performance, and area models, and iii) a set of area and performance constraints. The synthesis technique broadly consists of two stages: a initial solution generation stage followed by an iterative improvement stage.

4.4.1 Initial Solution

As a first step we generate an appropriate single-core SDF schedule. Our technique has an option to generate two different types of schedules: a minimum buffer schedule or a least switching schedule. For the minimum buffer schedule we utilize a well-known technique presented by Jantsch et al. [49]. A brief discussion on the generation of the schedules follows.

Table 4.1: Minimum Buffer PASS Generation Sequence

Step	Fired	Deferred	Non-Firable
1	A	A	B, B, C, C, D
2	B	A, C	B, C, D
3	C	A	B, C, D
4	A	—	B, C, D
5	B	C	D
6	C	—	D
7	D	—	—

Minimum Buffer Schedule: The first step in generating the schedule is to determine the number of times each actor must fire in order to maintain the buffer sizes (ie. prevent unrestricted buffer growth). This can be accomplished easily using firing vectors and solving a series of equations. Figure 4.5 illustrates a simple SDF specification. From the figure we can determine that actors A, B, and C must fire twice, while actor D must fire one time in order to maintain the initial buffer sizes. The next step in the generation of the minimum buffer schedule is to fire the next available actor that will increase the buffer requirement the least. To do this a table of the actors that are fired, deferred, and non-firable is maintained. An actor that is fired is the next actor in the schedule. A deferred actor is an actor that can be fired but has had it's firing delayed due to a better (smaller buffer increase) choice being available. Non-firable actors are actors that are unable to fire. Table 4.1 illustrates the process of firing the actors for the simple SDF in Figure 4.5 in order to keep the buffer growth to a minimum. This sequence of actor firings results in a maximum buffer requirement of 10 units and 6 switches between actors.

Least Switching Schedule: The first step in generating the least switching schedule is to determine the number of times each actor must fire in order

Table 4.2: Least Switching PASS Generation Sequence

Step	Fired	Deferred	Non-Firable
1	A	A	B, B, C, C, D
2	A	B, C	B, C, D
3	B	B, C, C	D
4	B	C, C	D
5	C	C	D
6	C	—	D
7	D	—	—

to maintain the buffer sizes. This is accomplished in the same way as with the minimum buffer schedule. Therefore, for the SDF in Figure 4.5 the actors A, B, and C must fire twice and actor D must fire one time. The next step is to fire the actors in such a manner that the number of times we switch between differing actors is minimal. To do this, we begin by firing the available actor that increases the buffer usage the least. We then see if the same actor can fire again. If it can, we fire it. If it can not, we fire a different available actor that will increase the buffer usage the least. Table 4.2 illustrates the process of firing actors in order to keep the actor switching to a minimum. This sequence of actor firings results in a maximum buffer requirement of 16 units and 3 switches between actors.

Each schedule type has advantages and disadvantages. The minimum buffer schedule is useful when the data blocks of the SDF are large and the execution time of the filters is long enough to hide the overhead of DMAing the filters in and out of main memory when code overlay schemes are in place. Least switching schedules are useful when the data blocks are small and the dominating aspect is the code size. Least switching schedules allow fewer fetches of code from main memory when code overlay schemes are in place.

This reduces the time overhead to fetch the code as well as the large increase to power consumption from accessing main memory.

After the multi-processor schedule is determined the next step in generating the initial solution is to perform code memory minimization. During this step we implement a code overlay scheme to reduce the memory requirement. In implementing the code overlay we attempt to keep the hit to performance to a minimal by only overlaying code blocks that have no interference. The next step is the data memory minimization step. During this step we implement an overlay scheme for the data blocks. Again we attempt to keep the impact on performance to a minimal by only overlaying data blocks that do not interfere with each other at all. After the completion of the code and data overlay steps we calculate the memory requirement of the code and data and generate a scratchpad memory (SPM) for the core. We also calculate the performance, area, and energy of the core utilizing the highest frequency (performance) software processing element available in the the library. This represents one software core. We then replicate this software core as many times as will fit in the area constraint. This will represent the initial solution. By beginning the iterative stage of our technique with a pure software design we are able to ensure that priority is given to maintaining as much functionality in software as possible. This is done by replacing filters in software with hardware accelerators only when necessary to meet performance and area constraints.

4.4.2 Data Memory Minimization

The objective of the data memory minimization stage is to create an overlay scheme to reduce the memory required for the data blocks by analyzing their lifetimes and assigning them to the same memory region. This is done while

```
1: $perform\_data\_overlay()$
2: {
3: $G = generate\_interference\_graph()$
4: while $\exists$ edge $(a, b) \in G$ whose I.F. $= 0$ do
5:    assign $a$ and $b$ to the same region.
6:    $update\_interference\_graph()$
7: end while
8: }
```

Figure 4.6: Data Memory Minimization Pseudo-code

attempting to minimize the impact on performance. The pseudo-code for the data minimization stage is shown in Figure 4.6. The first step is to generate the data block interference graph (Line 3, Figure 4.6). The interference graph is specified as a graph $G(V, E)$ where V is the set of data blocks and E is the set of edges from (u, v) where u and v are vertices in V. An edge (u, v) is annotated with the interference factor for the vertices u and v. The interference factor can be one of three values, i) 0, denoting the data blocks do not interfere in any way, ii) 1, denoting the data blocks interfere by being alive consecutively, and iii) 2, denoting the data blocks interfere by being alive at the same time. A data block is alive during the time frame it is being written to and again during the time frame it is being read from. Next, we look and see if there are any edges with zero interference (Line 4, Figure 4.6). If there is, we assign the two data blocks to the same region (Line 5, Figure 4.6). We then update the interference graph to include the combined interference of the newly overlayed data blocks (Line 6, Figure 4.6). This process continues until there no longer exists a pair of data blocks with an interference factor of zero.

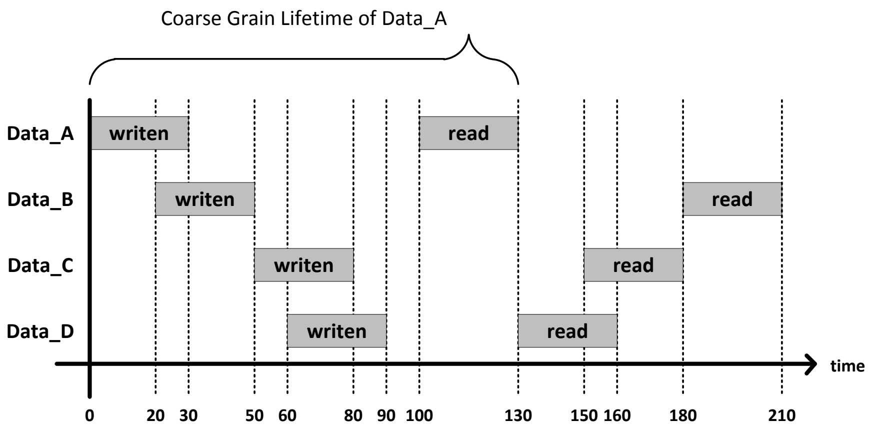


Figure 4.7: Example Data Lifetimes

Table 4.3: Data Inteference Table

—	A	B	C	D
A	—	2	0	1
B	2	—	1	0
C	0	1	—	2
D	1	0	2	—

Figure 4.7 illustrates a simple example with four data blocks A, B, C, and D along with their associated lifetimes. After analyzing the data block lifetimes we generate the interference graph (shown in table form) in Table 4.3. From the table we can see that data block pairs (A, C) and (B, D) both have interference factors of zero. Therefore, we would combine data blocks (A, C) into region 1 and data blocks (B, D) into region 2 and perform data overlay on the regions. Figure 4.8 illustrates the resulting schedule with the data blocks (A, C) and (B, D) being overlayed.

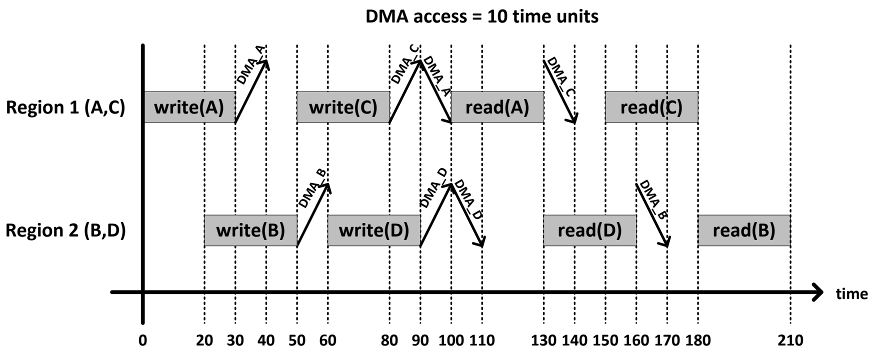


Figure 4.8: Data Overlay Schedule

4.4.3 Code Memory Minimization

The objective of the code memory minimization stage is to reduce the amount of memory required for the code by establishing a code overlay scheme. The pseudo-code for the code minimization stage is given in Figure 4.9. We begin by initializing each filter to its own region in memory (Line 3, Figure 4.9). Next we generate an interference graph (Line 4, Figure 4.9). The interference graph is specified as a graph $G(V, E)$ where V is the set of filters and E is the set of edges from (u, v) where u and v are vertices in V. An edge (u, v) is annotated with the interference factor of vertices u and v. The interference factor is defined as the number of times a consecutive transition is made from filter u to filter v and vice versa. Next, we check to see if there exists a pair of filters (regions) with an interference factor of zero (Line 5, Figure 4.9). If there is a pair we combine the regions together and update the interference graph (Lines 6 and 7, Figure 4.9). This process continues until there no longer exists a pair of filters (regions) with an interference factor of zero.

```
perform_code_overlay()
{
Initialize each filter to occupy its own region
$G = generate\_interference\_graph()$
while $\exists$ edge $(a, b) \in G$ whose I.F. $= 0$ do
    combine_regions()
    update_IF()
end while
}
```

Figure 4.9: Code Memory Minimization Pseudo-code

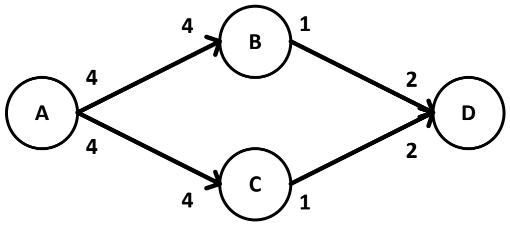


Figure 4.10: Example SDF Specification

Table 4.4: Code Inteference Table

—	A	B	C	D
A	—	1	0	1
B	1	—	1	0
C	0	1	—	1
D	1	0	1	—

Figure 4.10 illustrates a simple SDF specification. In the figure there are four filters (A, B, C, D). We will assume we are utilizing the least switching schedule A, A, B, B, C, C, D. Table 4.4 illustrates the interference graph (shown as a table) for the code filters. From the table we can see that filters (A, C) and filters (B, D) have interference factors of zero. Therefore, we would

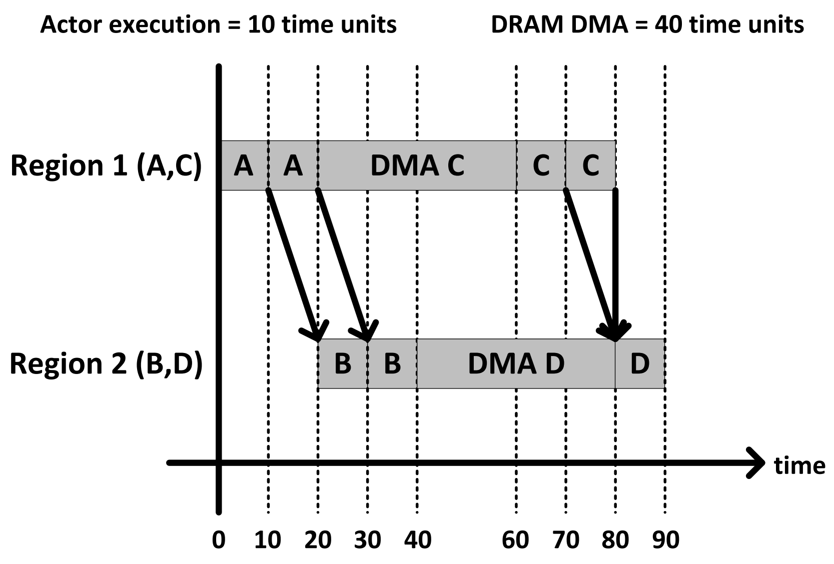


Figure 4.11: Least Switching Schedule

```
1: $hardware\_accelerator\_replication()$
2: {
3: for each hardware accelerator $X$ do
4:     for i = 2; i < number of SW core replicates; i++ do
5:         create new hardware accelerator with attributes $X * i$
6:     end for
7: end for
8: }
```

Figure 4.12: Hardware Accelerator Replication Pseudo-code

combine filters (A, C) into region 1 and filters (B, D) into region 2. Figure 4.11 illustrates the resulting execution schedule with DMAs. From the illustration and the assumed execution times and DMA time, we can see that the overhead of the DMA is partially hidden by the execution of the filters in the other region.

Table 4.5: Hardware Accelerators Before Replication

Hardware Accelerator	Area (mm^2)	Performance μs	Energy (nJ)
1	.29	5.64	26112.90
2	.45	1.16	31312.32
3	1.69	0.04	36372.78

4.4.4 Hardware Accelerator Replication

The hardware accelerator replication stage allows us to expand the library of hardware accelerator processing elements to include processing elements with performance, area, and power models that would otherwise be absent. Thus, giving us a more complete library to choose from when transitioning from the pure software initial solution to the HW-SW hybrid solution. The pseudo-code for the hardware replication stage is shown in Figure 4.12. We begin by iterating through all of the initial hardware accelerators (Line 3, Figure 4.12). Next, we iterate from 2 to the total number of software cores in the initial pure software solution (Line 4, Figure 4.12). For each value of i we create a new hardware accelerator with attributes equal to replicating the hardware accelerator i times (Line 5, Figure 4.12). This continues until every hardware accelerator has been replicated fully.

Tables 4.5 and 4.6 show a simple hardware accelerator library before and after hardware replication, respectively. Each hardware accelerator was replicated twice to form a new accelerator (denoted by #.2). From the figures we see through replication we are able to provide new hardware accelerators that fill the gaps that would otherwise be absent.

Table 4.6: Hardware Accelerators After Replication

Hardware Accelerator	Area (mm^2)	Performance μs	Energy (nJ)
1	.29	5.64	13112.90
1.2	.58	2.82	26225.80
2	.45	1.16	31312.32
2.2	.90	0.58	62624.64
3	1.69	0.04	66372.78
3.2	3.38	0.02	132745.56

```
1: $move\_filter\_to\_hw()$
2: {
3: while performance constraint not met && SW still exists do
4:    $find\_slowest\_filter()$
5:    $remove\_filter\_from\_SW()$
6:    $add\_hw\_double\_buffering()$
7:    while $add\_HW\_accelerator()$ == FAIL do
8:       $remove\_SW\_core()$
9:       if number SW cores == 0 then
10:          $build\_hw\_design()$
11:       end if
12:    end while
13: end while
14: if performance is met && area is met then
15:    $reduce\_power()$
16: else
17:    output failure
18: end if
19: }
```

Figure 4.13: Iterative Transition to HW-SW Design Pseudo-code

4.4.5 Iterative Transition to HW-SW Design

During the iterative transition to a hw-sw design stage of the technique we incrementally move a filter from being executed in software to a dedicated hardware accelerator in order to improve the performance of the design while maintaining a similar area requirement. Typically we will see a reduction in the power consumption during this stage due to hardware requiring less power and the reduction of filter code being fetched from DRAM. The pseudo-code for the transition from a pure SW design to a HW-SW co-design is shown in Figure 4.13.

The code begins by checking to see if the performance constraint is met and whether there exists any filters remaining in software (Line 3, Figure 4.13). Next, we find the slowest filter in software (Line 4, Figure 4.13). This filter will be the filter that we move from software to a hardware accelerator. In the next step, we remove the filter from the software core(s) (Line 5, Figure 4.13). To do this, we perform three tasks: i) we remove the filter from the software execution schedule, ii) we remove the filter data blocks from memory and update the data overlay scheme if applicable, and iii) we remove the filter code block from memory and update the code overlay scheme if applicable. By maintaining the previously generated interference graphs we can quickly and easily determine if by removing the filter from software whether we can create additional overlays for code and data in the memory of the software core(s).

The next stage of the algorithm is to add the hardware accelerator for the removed software filter. To do this we first add double buffering to the software core(s) for the data to/from the hardware accelerator (Line 6, Figure 4.13). By adding double buffering the software and hardware can execute in parallel. Next, we attempt to add the hardware accelerator to the design (Line 7, Figure 4.13). In order for the hardware accelerator to be successfully added it must, i) fit in the available area, ii) execute $n * i$ times within the performance constraint, where n is the number of software cores and i is the number of instances the filter exists in the execution schedule. If there does not exist a hardware accelerator capable of meeting these requirements we remove one software core and try again (Line 8, Figure 4.13). However, if we remove the last software core we build a pure hardware design (Line 10, Figure 4.13).

To build a pure hardware design we allocate the slowest hardware accelerator for each filter in the SDF. We then iteratively improve the performance of the slowest filter, by changing it to a faster accelerator, until the performance constraint is met or the area constraint will become violated. This approach, when successful, will yield a pure hardware solution with the minimal area and energy for the given performance constraint.

If the algorithm is able to generate a design that meets both the area and performance constraints it will try to reduce the energy consumption (Line 14 and 15, Figure 4.13). To do this, the faster processing elements (hardware and software) are one-at-a-time switched out for slower more energy conscious alternatives. This continues until no processing elements exist that can be switched out without violating the performance constraint.

If the algorithm is unable to meet the performance and area constraint, it will output a failure (Line 17, Figure 4.13).

Table 4.7: Benchmark Specifications

Benchmarks	#Actors	#Edges	#Executions
Beamformer	40	72	64
Bitonic-sort	26	31	68
Channelvocoder	55	70	87
DCT	15	22	28
FFT	17	16	58
Filterbank	51	65	94
Fmradio	29	39	58
Average	33	45	65

4.5 Experimental Results

We evaluated the efficacy of our approach through the use of seven benchmarks from the StreamIT [44] benchmark suite. The benchmarks are described in Table 4.7. In the table the second and third columns denote the number of actors and edges in each benchmark, and the last column denotes the total number of actor firings in one iteration of the SDF. We set the performance constraint to a set value for each benchmark and varied the area constraint. The constraints used for each benchmark are shown in Table 4.8.

We compared our technique against four different initial solutions, i) one with both code and data overlay (denoted as "Code and Data Overlay" in the plots), ii) one with only code overlay (denoted as "Code Overlay" in the plots), iii) one with only data overlay (denoted as "Data Overlay" in the plots), and iv) one with neither code or data overlay (denoted as "No Overlay" in the plots). We also compared against a pure hardware solution (denoted as "Hardware" in the plots). Each initial software solution utilized the highest performing software core. The pure hardware designs were the smallest (lowest energy) solution that met the performance constraint. The solutions generated

Table 4.8: Benchmark Constraints

Benchmarks	Performance Constraint (μs)	Area Constraints (mm^2)
Beamformer	0.25	5,10,20,...,70
Bitonic-sort	0.33	1,5,10,15,20
Channelvocoder	120	2,6,11,17,22,30,35
DCT	2	1,5,10,15,20,25
FFT	10	10,50,100,150,200
Filterbank	50	1,5,10,20,...,110
Fmradio	40	5,10,15,20,30,35,45,50

by our technique utilized the initial solution with both code and data overlays. We generated designs for both a minimum buffer schedule and a minimum switching schedule. We compared the designs generated by each method in terms of performance vs. area vs. energy. We also analyzed the overall impact overlay schemes have on the initial solution.

4.5.1 High Level Synthesis of Hardware Accelerator Library

In order to generate the designs, we needed to generate the hardware accelerator library. To do this, we utilized the software implementations of the benchmark filters provided by the StreamIT [44] compiler. We converted the software (C/C++) implementations into SystemC [50] hardware descriptions. We then utilized the high-level synthesis tool Forte Cynthesizer [46] to synthesize the SystemC descriptions into hardware. By changing the synthesis constructs within the SystemC files we were able to synthesize several hardware accelerators for each filter with varying area and performance models. During the process of high-level synthesis the Cynthesizer tool outputs RTL code for the hardware accelerators. We used the produced RTL code to generate energy models using Synopsys Primepower [47]. During the process of high-level synthesis and the gathering of the energy values we utilized the TSMC 45nm libraries from Synopsys. Due to the extensive manual labor required to generate the hardware accelerators we only performed the high-level synthesis tasks for two of the benchmarks: DCT and FFT. For the other benchmarks, we performed estimations for the performance, area, and energy models of the filters based on the results generated for the filters of the DCT and FFT benchmarks.

The software cores utilized in our results are from the ARM Cortex R4 [48] series of cores. The memory library (SPM and DRAM models) was generated based on the findings in Banakar, et al. [45]. The capacity of the SPM memories were set to power of two increments ranging from 32B to 524KB.

Due to the large quantity of results, we will only discuss in detail the results for the DCT and FFT benchmarks in this section. The results for the additional benchmarks will be presented in Section 4.6 with minimal discussion.

4.5.2 DCT Comparison

In this section we will analyze the results for the DCT benchmark. Figures 4.14 and 4.15 illustrate the results for the performance vs. area vs. energy comparison when the smallest buffer execution schedule is utilized. In the figures the light gray dashed line represents the performance constraint. From the figure we can see that the software solution with code and data overlay is only able to meet the performance constraint when the area constraint is quite

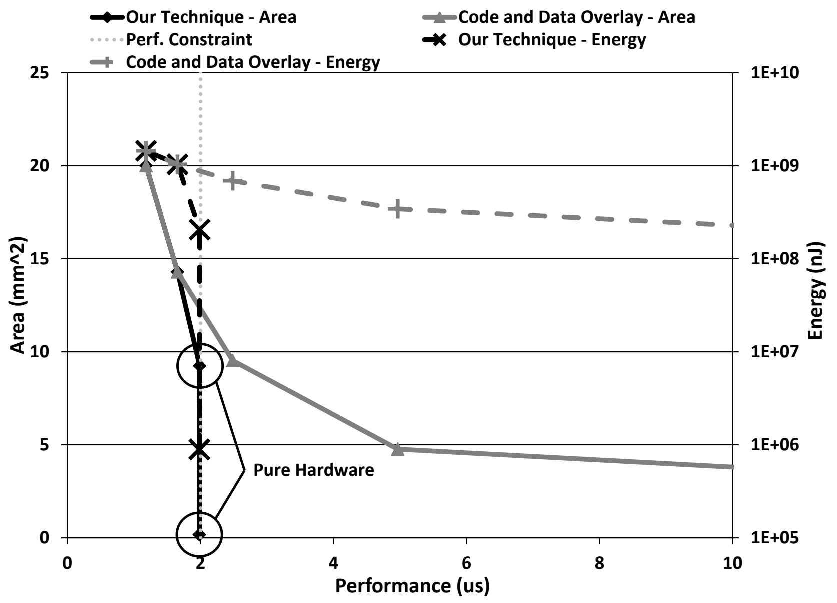


Figure 4.14: DCT Area vs. Perf. vs. Energy (sb)

large. However, the software solution consumes a large amount of energy. The high energy consumption can be contributed to two main causes, 1) the higher cost to operate a software core compared with a hardware accelerator, and 2) the high cost of accessing the main memory. From the figure, we can see that the designs generated by our technique are always capable of meeting the area and performance constraints. The designs we generate place a priority on software cores, only eliminating them when necessary. This is evident in the figure by the plot representing our designs ("Our Technique") following the pure software designs until hardware accelerators are required. At which point, our technique generates HW-SW designs which meet the performance constraint while maintaining as much functionality in software as possible. Our technique only resorts to a pure hardare design (denoted by circles in the

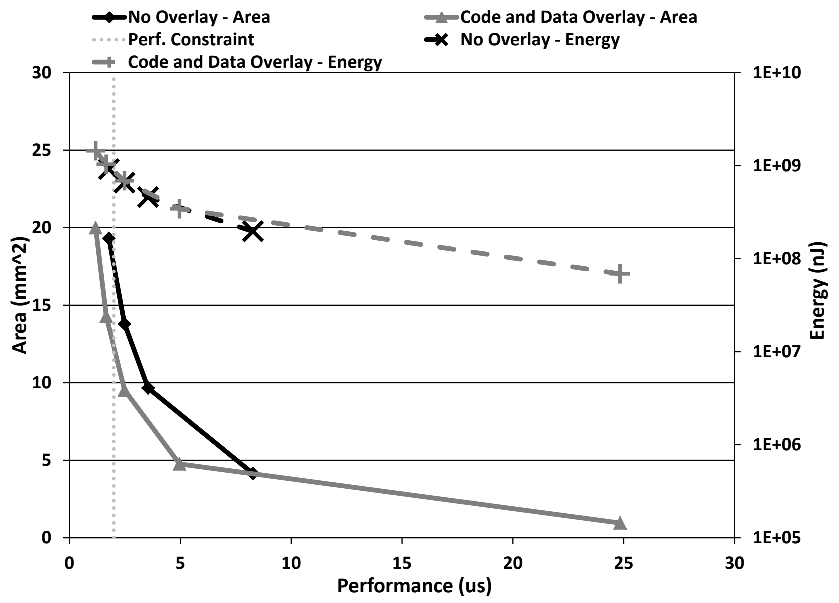


Figure 4.15: DCT Area vs. Perf. vs. Energy (sb)

figure) when the area constraint is too tight that a HW-SW design is infeasible. From the figure we can see as the designs generated by our technique transition from a pure software design towards a pure hardware design the energy consumption drastically decreases. This is expected due to the reduction in main memory accesses as more and more functionality of the application is being moved onto hardware accelerators. From Figure 4.15 we can see that by using code and data overlay we are able to generate valid solutions for the pure software designs longer. However, this comes with a cost of the energy consumption increasing from the use of code and data overlays. Again, this is expected due to more main memory accesses to retrieve code and data.

Figures 4.16 and 4.17 illustrate the results for the performance vs. area

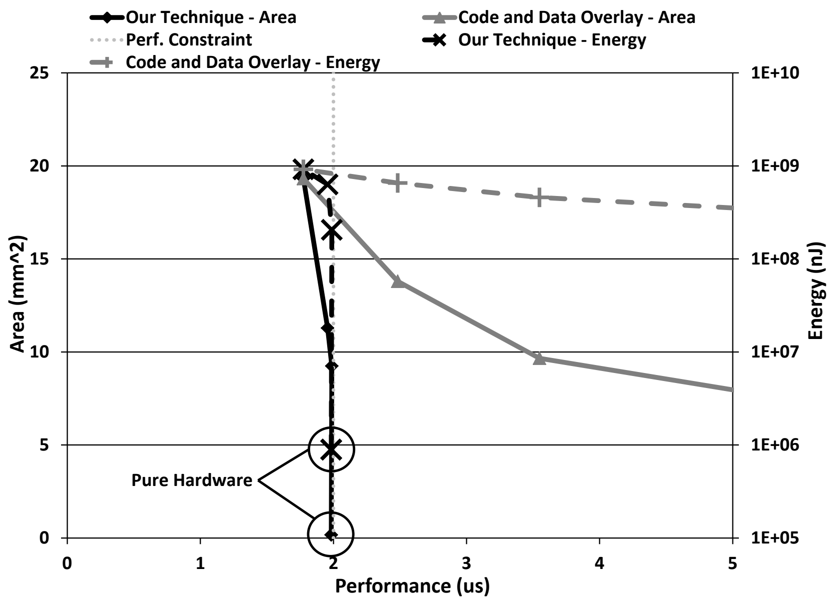


Figure 4.16: DCT Area vs. Perf. vs. Energy (ls)

vs. energy comparison when the least switching execution schedule is utilized. From the figures we can see a similar trend as with the previous figures. However, one thing to note is due to the least switching schedule the software designs do not improve with the use of code or data overlay. Because of this the software designs are unable to meet the area and performance constraints much sooner. Therefore, forcing our technique to generate HW-SW designs and ultimately a pure hardware design as the area constraint is tightened.

4.5.3 FFT Comparison

In this section we will analyze the results for the FFT benchmark. Figures 4.18 and 4.19 illustrate the results for the performance vs. area vs. energy comparison when the smallest buffer execution schedule is used. From the

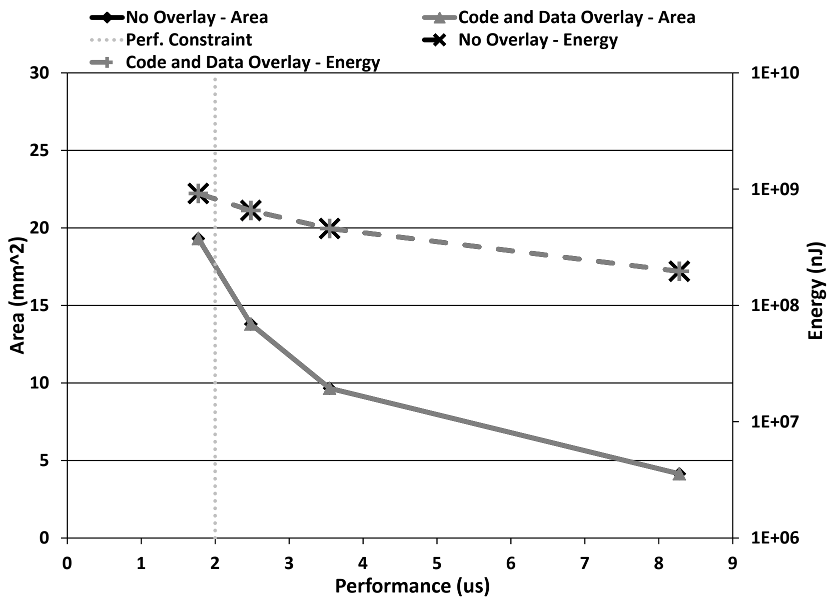


Figure 4.17: DCT Area vs. Perf. vs. Energy (ls)

figures we can see that the plots are similar to those for the DCT benchmark. From Figure 4.18 we can see that the designs generated by our technique follow the pure software designs until the designs no longer meet both the area and performance constraint. At this point, our technique begins to generate HW-SW designs with a priority on maintaining as much functionality in software as possible. This evident in the figure by the plot for our technique ("Our Technique") following the performance constraint line rather than jumping directly to a pure hardware design. Ultimately, when the area constraint is restricted far enough our technique is forced to generate a pure hardware design in order to meet the constraints. These pure hardware designs are denoted in the figure by the circles. The figure also illustrates that the energy consumption for the pure software designs is significantly higher than the

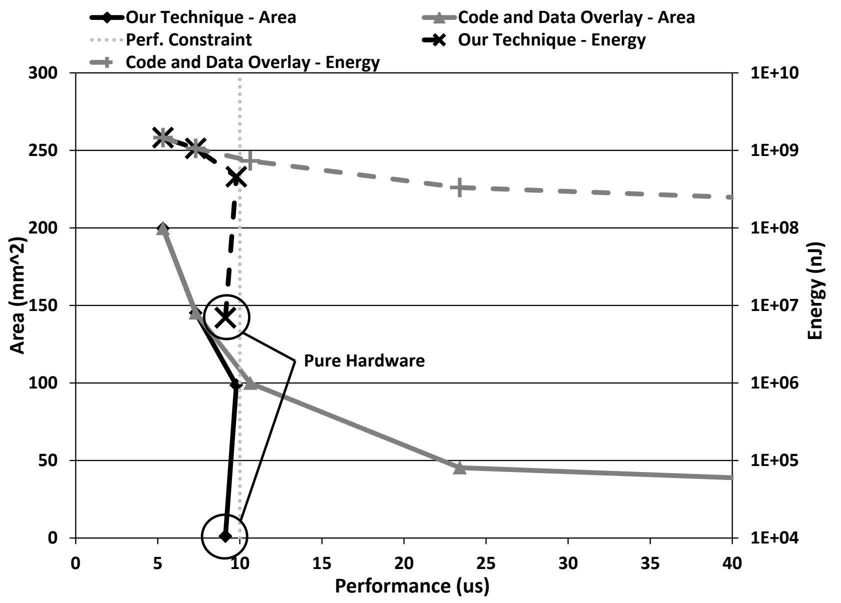


Figure 4.18: FFT Area vs. Perf. vs. Energy (sb)

energy consumption for the HW-SW designs generated by our technique as well as the pure hardware design. This is anticipated due to the high cost of accessing the main memory along with the higher energy cost to operate a software core versus a hardware accelerator. Figure 4.19 illustrates a similar trend as Figure 4.15. In the figure we can see that by using code and data overaly we are able to generate pure software solutions that meet both the area and performance constraint longer. However, this comes at a cost of higher energy consumption. The increase in energy consumption is associated with the code and data overlay schemes requiring accesses to the main memory.

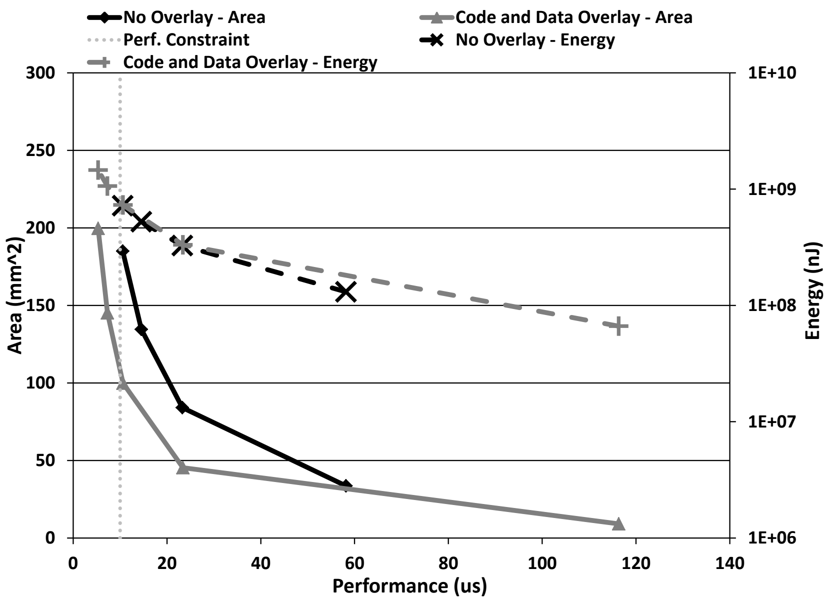


Figure 4.19: FFT Area vs. Perf. vs. Energy (sb)

Figures 4.20 and 4.21 illustrate the results for the performance vs. area vs. energy comparison when the least switching execution schedule is used. From Figure 4.20 we can see that the pure software designs are never able to meet the area and performance constraint. Therefore, forcing our technique to generate HW-SW designs until the area constraint is tight enough to require a pure hardware design (denoted by the circles). Figure 4.21 illustrates that with the least switching execution schedule the pure software designs are unable to take advantage of code and data overlays in order to reduce the footprint of each software core. By reducing the footprint of each software core we are able to place more software cores in the same area. Thus, increasing the performance of the design.

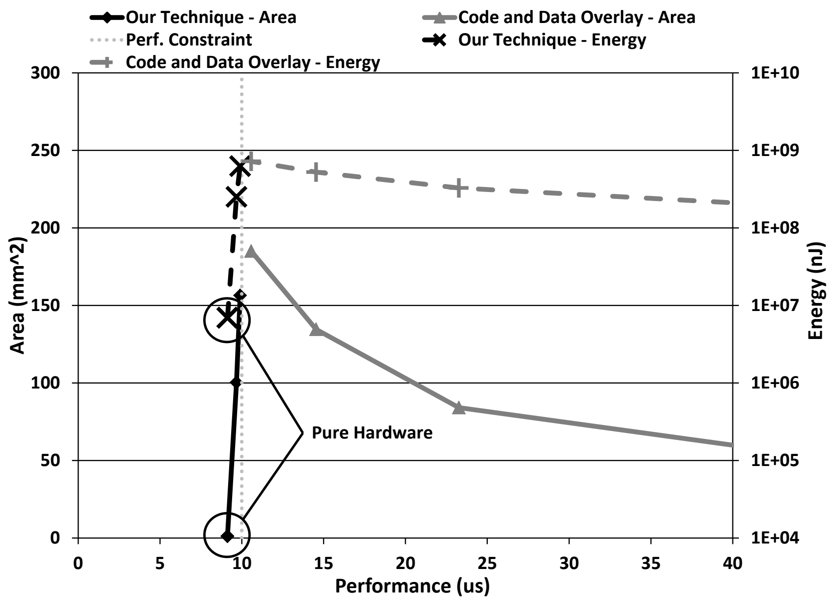


Figure 4.20: FFT Area vs. Perf. vs. Energy (ls)

4.5.4 Impact of Overlay

In this section we will discuss the impact of using code and data overlay on the pure software solution. Since, we use a pure software solution as the starting point in our technique it is worthwhile to discuss the impact of overlay schemes in further detail. Figures 4.22 and 4.23 illustrate the impact of using overlay schemes on the DCT benchmark when a smallest buffer execution schedule and a least switching execution schedule are used, respectively. In the figures the light gray dashed line denoted with "Perf. Constraint" represents the performance constraint used for the DCT benchmark. From Figure 4.22 we can see that with no overlay scheme as well as with code overlay we generate

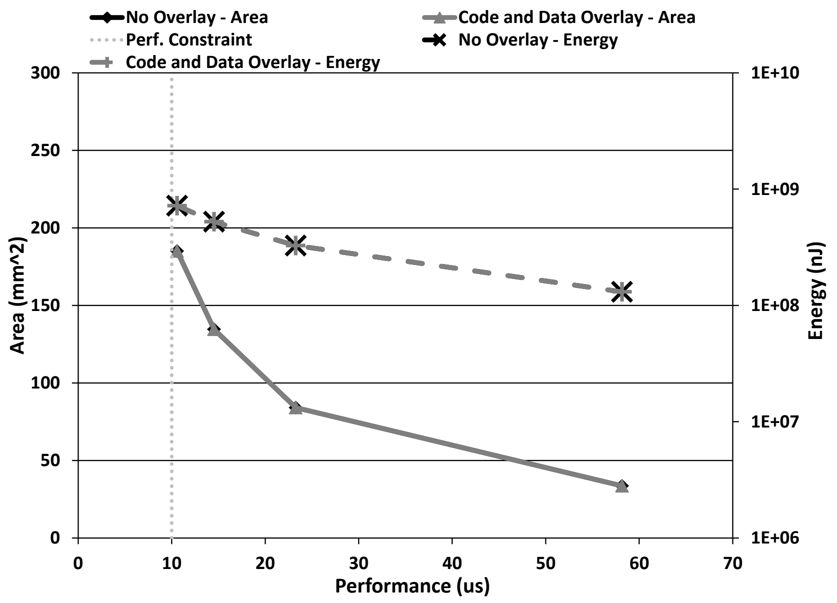


Figure 4.21: FFT Area vs. Perf. vs. Energy (ls)

the same designs in terms of area and performance. This does not mean that code overlay is not being performed. What this represents is that with code overlay we are unable to use a smaller scratchpad memory (SPM) size. Therefore, the area requirement remains the same. Further, the use of code overlay is not negatively impacting the performance of the design. If we recall, not decreasing performance was a priority when generting overlay schemes. From Figure 4.22 we can see that using only data overlays we are able to improve the performance of the design. This is due to data overlays allowing us to use a smaller SPM and therefore add more software cores. Thus, improving the performance of the design. Lastly, from the figure we can see when we implement both code and data overlays we are able to achieve the best performance. This is again due to the savings in area by using a smaller SPM to

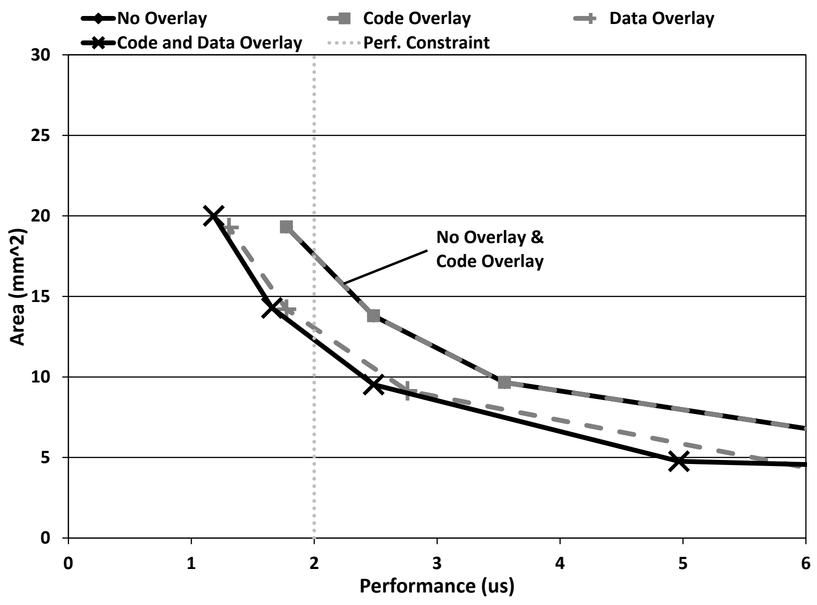


Figure 4.22: DCT Overlay Impact (sb)

store the code and data, which allows more software cores to be allocated to the design. Thus, improving the performance of the design.

Figure 4.23 illustrates an interesting concept when all four software designs are the same. This occurs when the execution schedule (in this case the least switching schedule) does not allow for any significant code or data overlays to be implemented. Therefore, we are unable to save area through overlays and thus unable to allocate additional software cores in order to improve the performance of the design.

Lastly, Table 4.9 illustrates the performance, area, and energy values of a single software core for each pure software design for the FFT benchmark when a smallest buffer execution schedule is used. From the table, we can

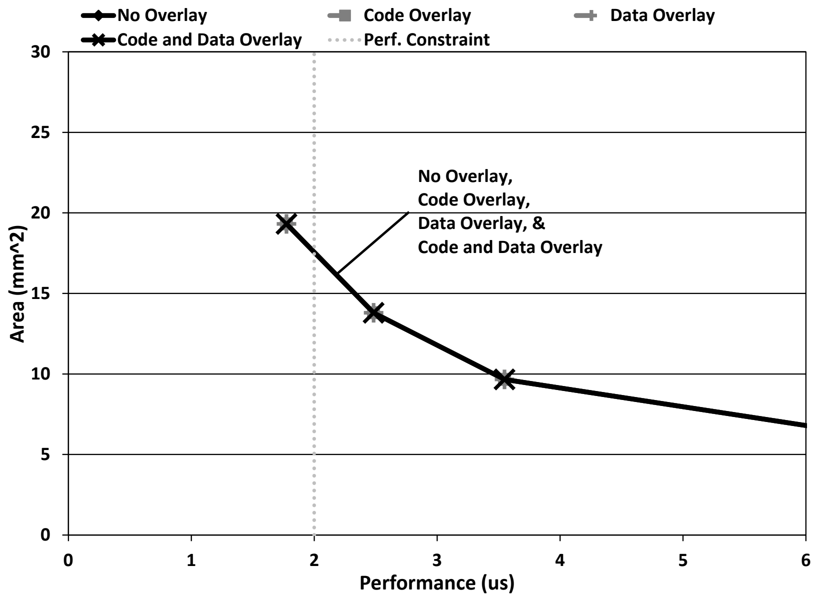


Figure 4.23: DCT Overlay Impact (ls)

Table 4.9: Single Software Core Comparison (FFT)

Overlay Type	Performance (μs)	Area (mm^2)	Energy (nJ)
No Overlay	116.316	16.8176	65482600
Code Overlay	116.998	16.8176	65549100
Data Overlay	116.316	9.074	66303000
Code and Data Overlay	116.998	9.074	66369600

see when we only implement code overlay we receive a software core with worse performance, equivalent area, and higher energy consumption as the software core with no overlays. The impact to performance and energy is due to fetching code from main memory. However, the thing to note is that the area requirement remained the same. This is due to the memory savings from code overlay not being substantial enough to allow the use of a smaller

SPM. However, if we implement only data overlays we see no decrease to performance, a reduction in area, and an increase to energy. The increase to energy is expected due to accessing the main memory. The reduction in area is contributed to the memory savings of overlaying data blocks allowing the core to utilize a smaller SPM. The lower area requirement through the use of data overlays would allow more software cores to be allocated in the same area constraint as the softwares core with no overlays and code overlays. Lastly, from the table we can see that the use of both code and data overlay results in a software core with decreased performance, smaller area requirement, and higher energy consumption than the software core with no overlays. From this analysis we can conclude that always using the software core with both code and data overlays may not be the best option. For instance, in this case the software core with data overlays only would be the best choice.

4.6 Extended Results

In this section we discuss the performance vs. area vs. energy results for the remainder of the benchmarks.

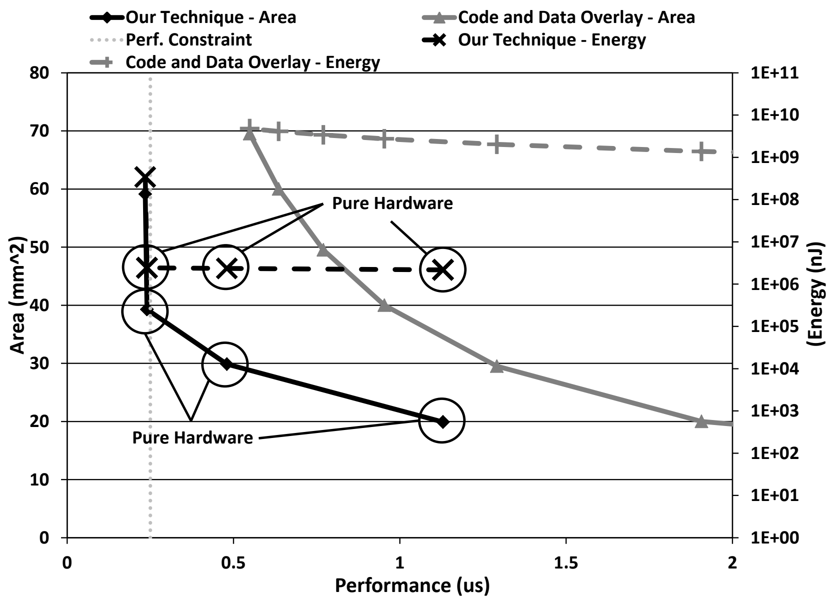


Figure 4.24: beamformer Area vs. Perf. vs. Energy (sb)

Figures 4.24 and 4.25 illustrate the performance vs. area vs. energy comparison when a smallest buffer execution schedule is used. From the figures we can see that the plots follow a similar trend as those discussed previously. However, one interesting point is that there are multiple pure hardware solutions in Figure 4.24. This occurs as the pure hardware design attempts to meet both the area and performance contraint. Until both constraints have been met the design will change as more area is made available in order to allocate higher performing hardware accelerators. Figures 4.26 and 4.27 illustrate the comparison when a least switching execution schedule is used. From the figures we can see that they follow a similar trend as previously discussed.

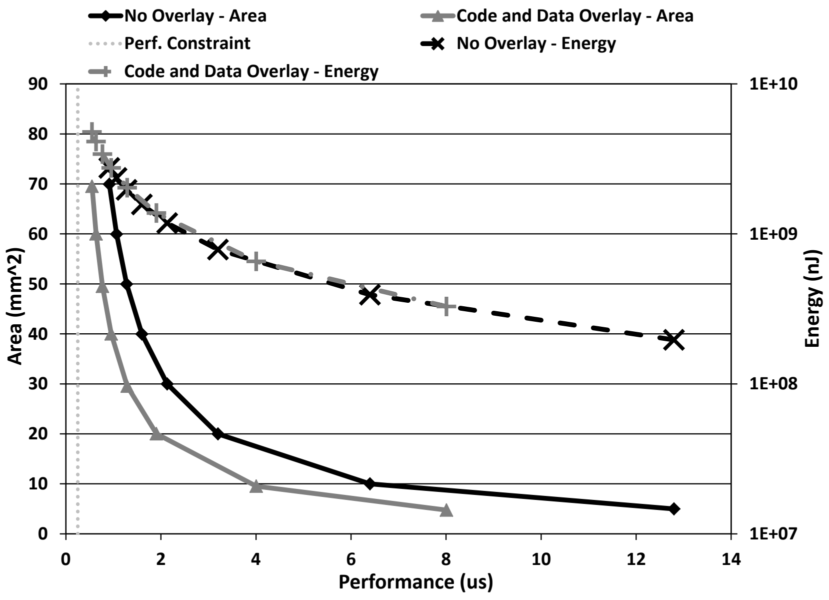


Figure 4.25: beamformer Area vs. Perf. vs. Energy (sb)

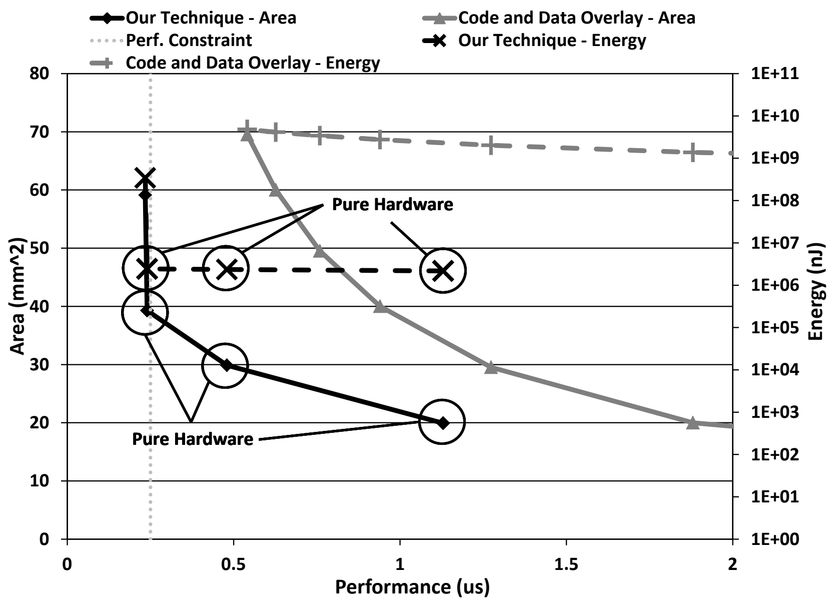


Figure 4.26: beamformer Area vs. Perf. vs. Energy (ls)

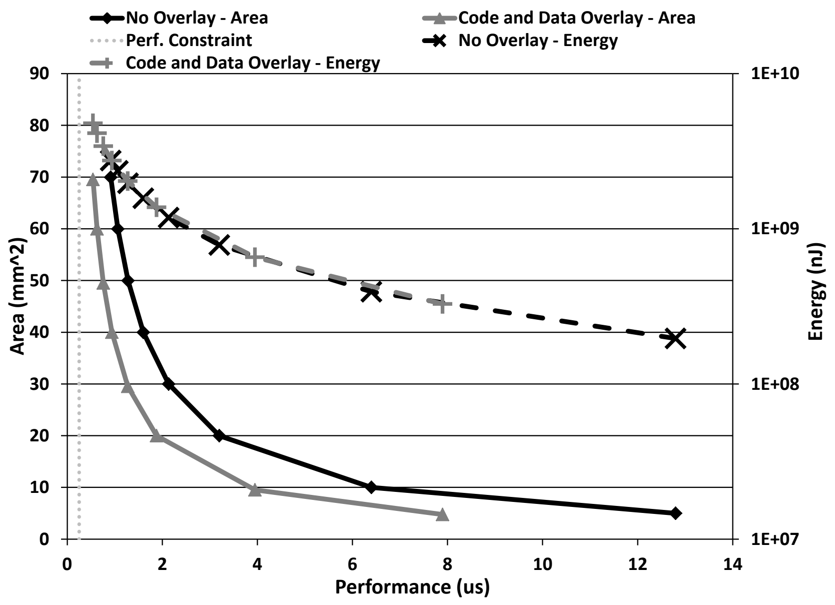


Figure 4.27: beamformer Area vs. Perf. vs. Energy (ls)

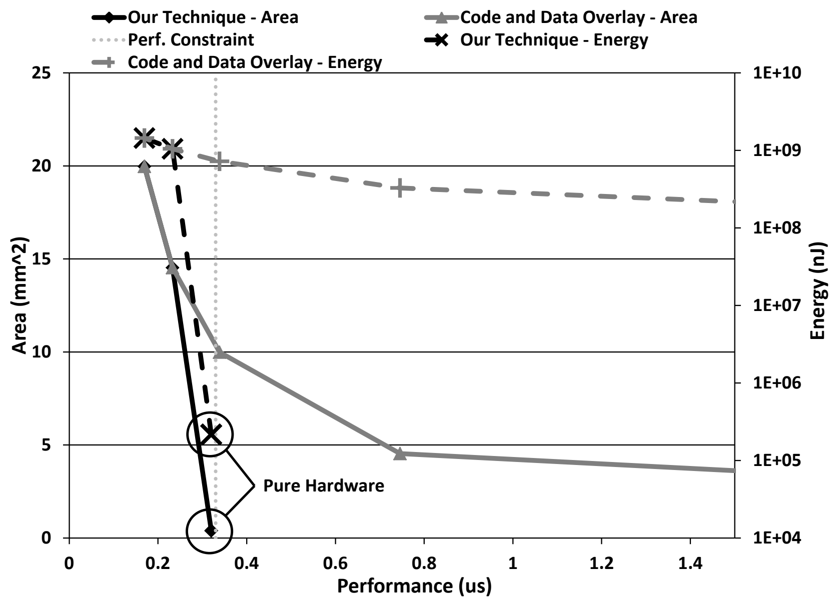


Figure 4.28: bitonic-sort Area vs. Perf. vs. Energy (sb)

Figures 4.28, 4.29, 4.30, and 4.31 illustrate the performance vs. area vs. energy comparison for the bitonic-sort benchmark when a smallest buffer execution schedule and a least switching execution schedule are used, respectively. From the figures we can see that the plots follow a similar trend as previously discussed. Our technique generated designs which are pure software until no longer feasible due to the area constraint. At which point, are technique will generate a HW-SW design until ultimately being forced to generate a pure hardware design.

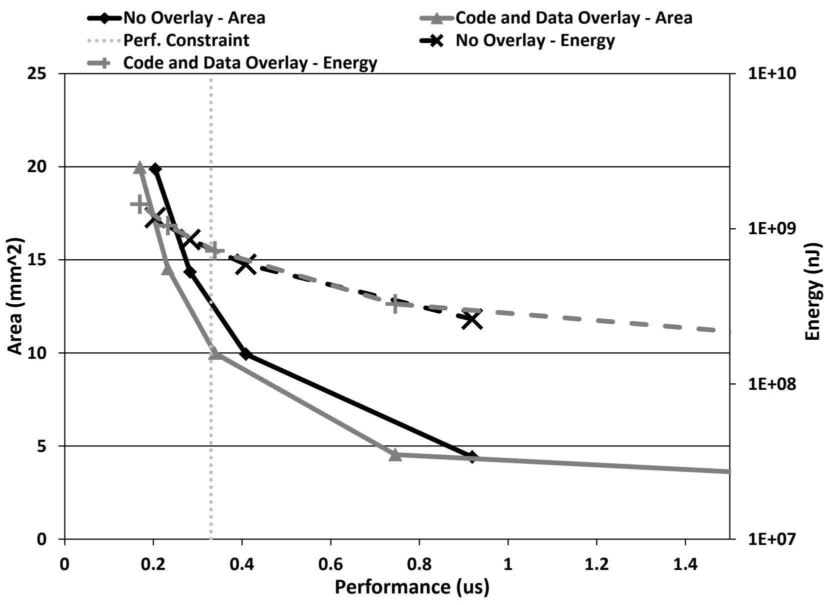


Figure 4.29: bitonic-sort Area vs. Perf. vs. Energy (sb)

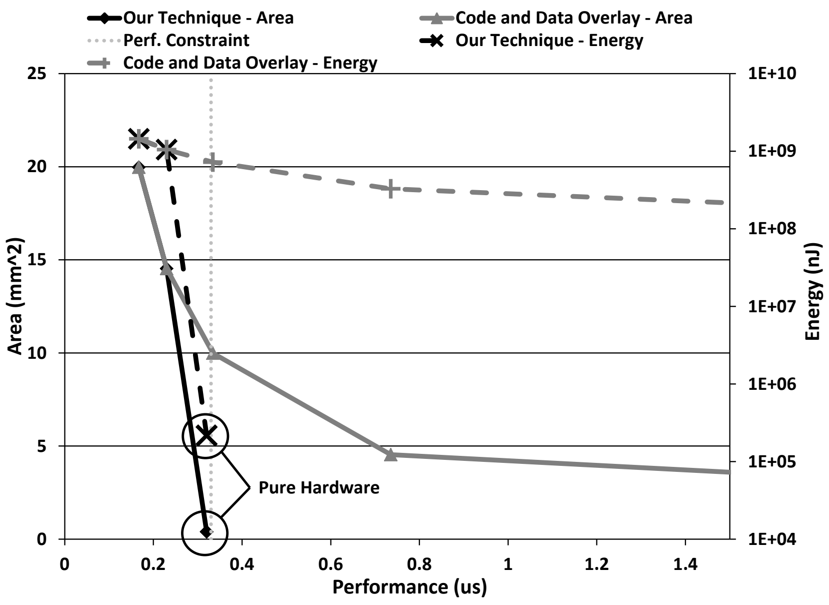


Figure 4.30: bitonic-sort Area vs. Perf. vs. Energy (ls)

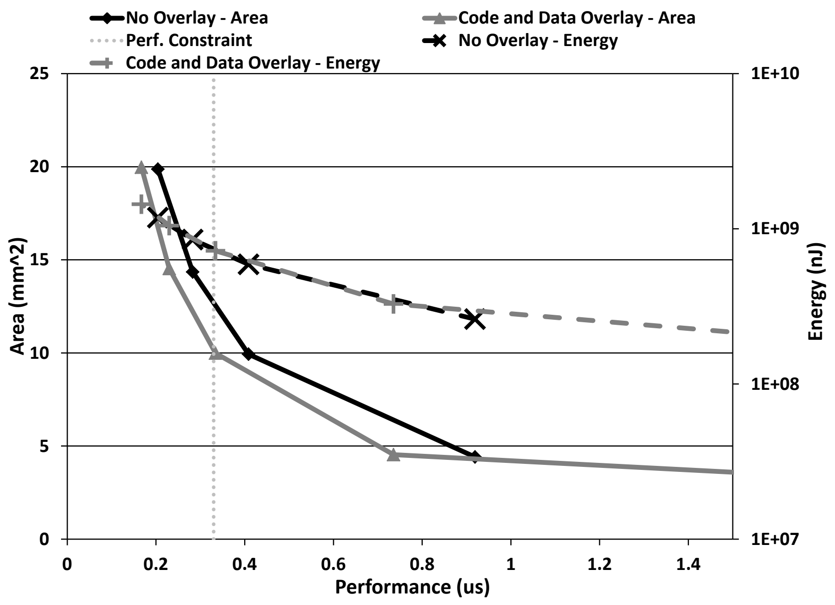


Figure 4.31: bitonic-sort Area vs. Perf. vs. Energy (ls)

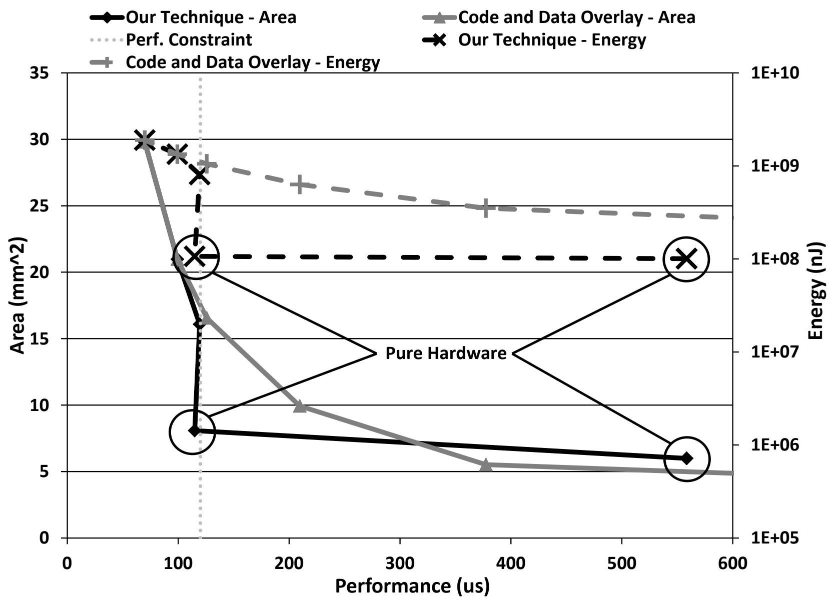


Figure 4.32: channelvocoder Area vs. Perf. vs. Energy (sb)

Figures 4.32, 4.33, 4.34, and 4.35 illustrate the performance vs. area vs. energy comparison for the channelvocoder benchmark when a smallest buffer execution schedule and a least switching execution schedule are used, respectively. From the figures we can see that the plots follow a similar trend as previously discussed.

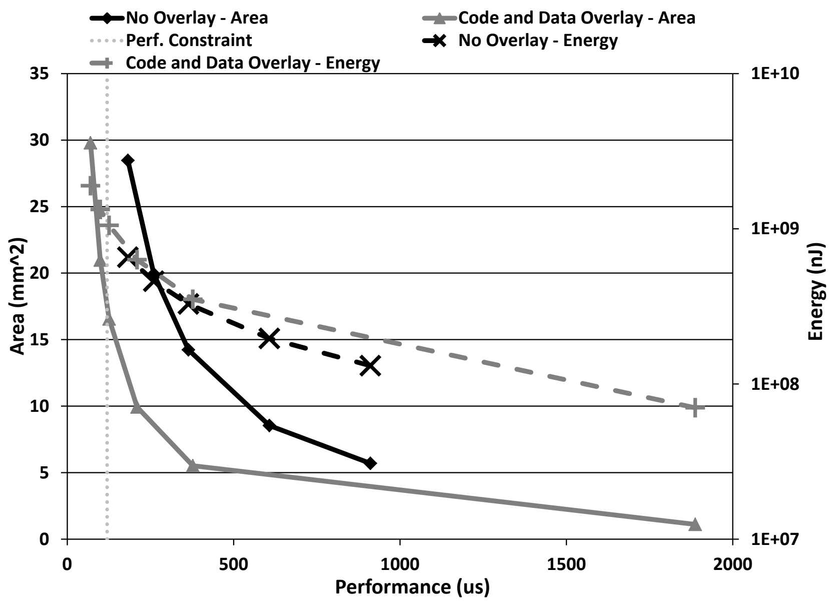


Figure 4.33: channelvocoder Area vs. Perf. vs. Energy (sb)

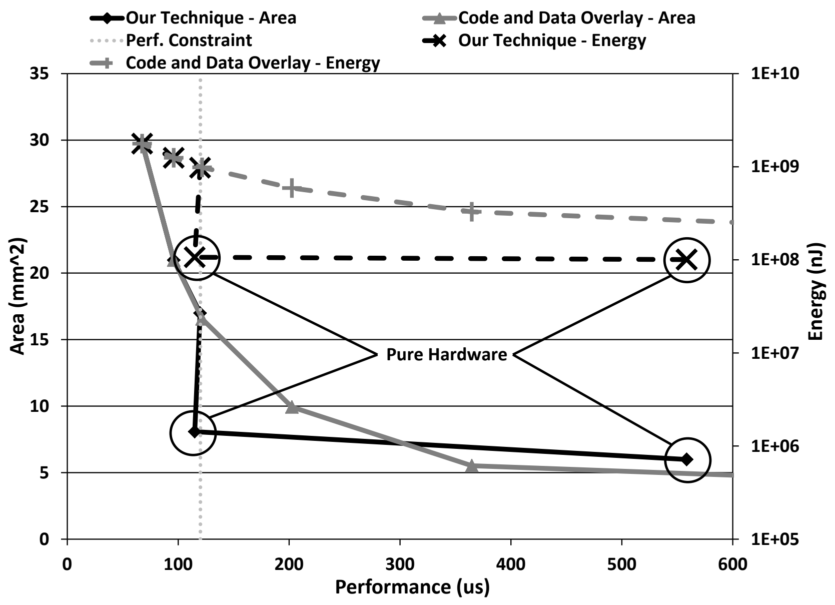


Figure 4.34: channelvocoder Area vs. Perf. vs. Energy (ls)

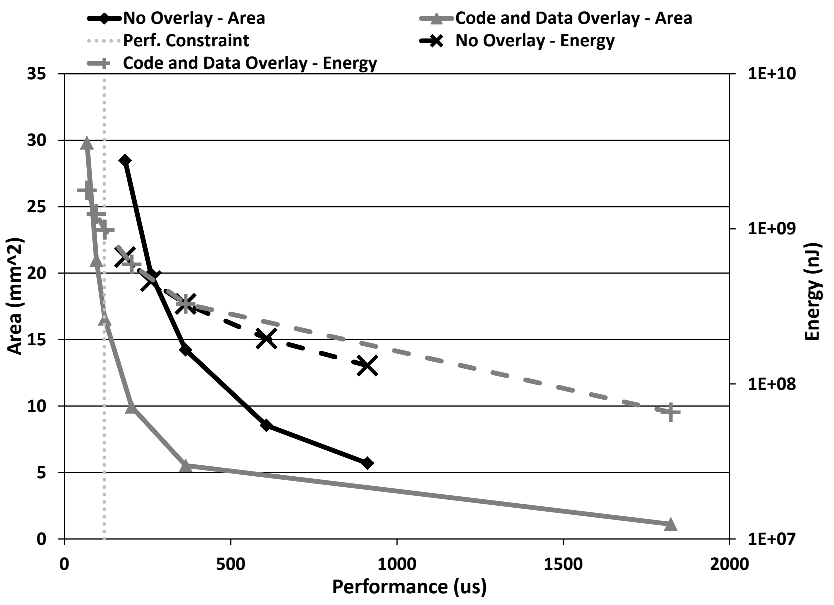


Figure 4.35: channelvocoder Area vs. Perf. vs. Energy (ls)

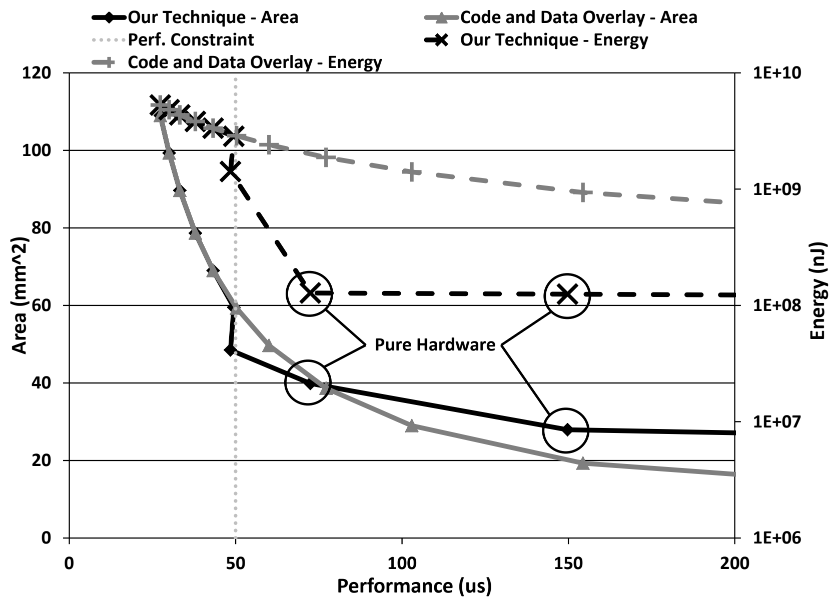


Figure 4.36: filterbank Area vs. Perf. vs. Energy (sb)

Figures 4.36, 4.37, 4.38, and 4.39 illustrate the performance vs. area vs. energy comparison for the filterbank benchmark when a smallest buffer execution schedule and a least switching execution schedule are used, respectively. From the figures we can see that the plots follow a similar trend as previously discussed.

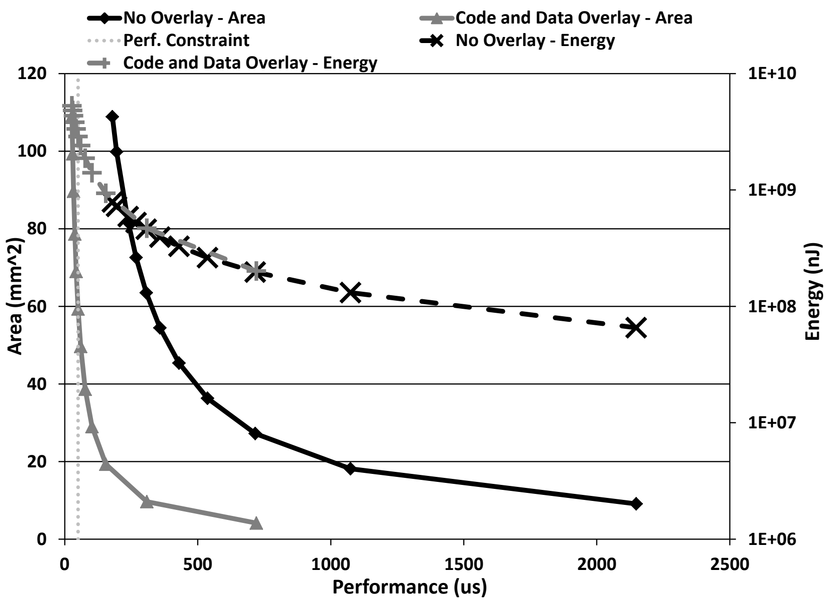


Figure 4.37: filterbank Area vs. Perf. vs. Energy (sb)

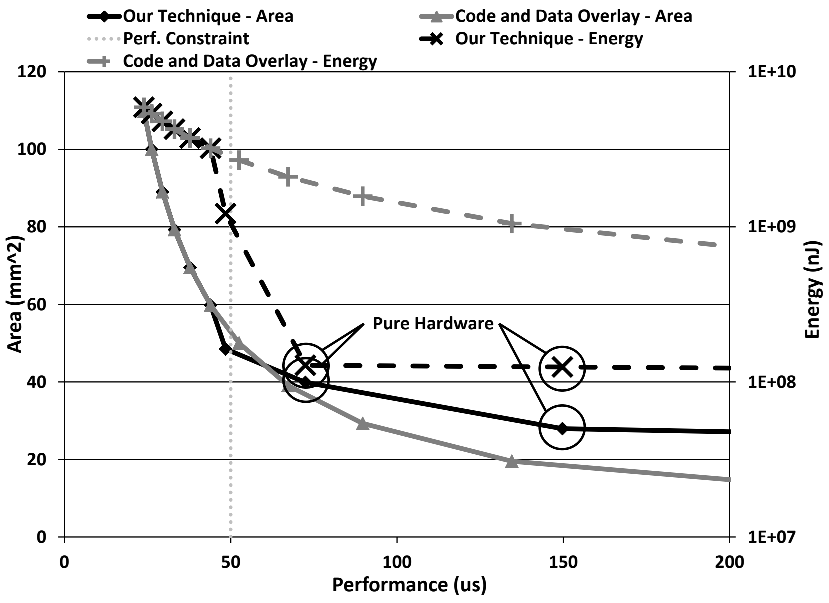


Figure 4.38: filterbank Area vs. Perf. vs. Energy (ls)

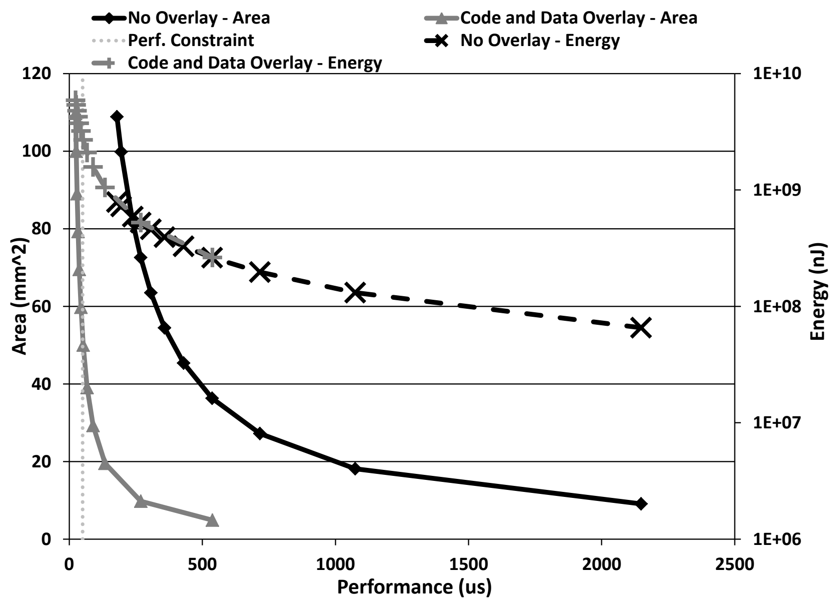


Figure 4.39: filterbank Area vs. Perf. vs. Energy (ls)

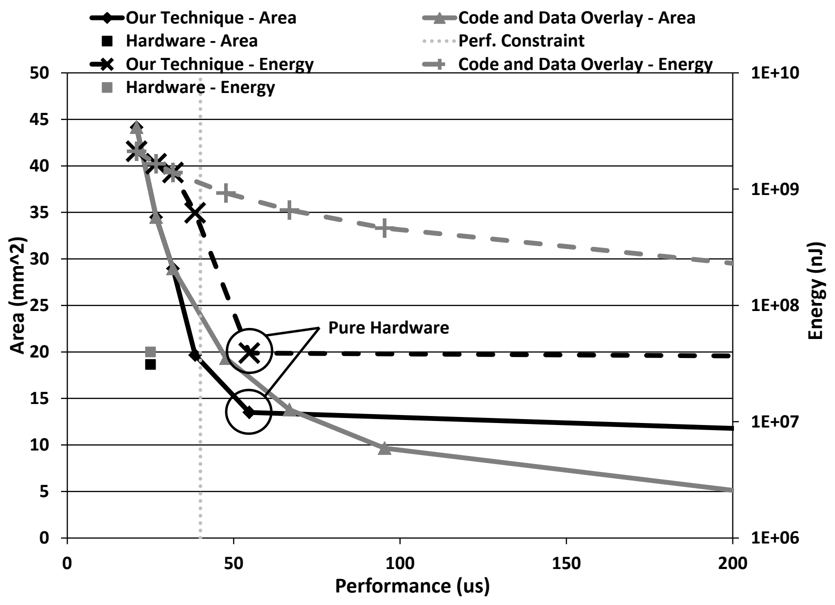


Figure 4.40: fm Area vs. Perf. vs. Energy (sb)

Figures 4.40, 4.41, 4.42, and 4.43 illustrate the performance vs. area vs. energy comparison for the fm benchmark when a smallest buffer execution schedule and a least switching execution schedule are used, respectively. From the figures we can see that the plots follow a similar trend as previously discussed. There is one point to note however. In Figures 4.40 and 4.42 our technique never generates a valid pure hardware design. We can see from the figures that the closest design we create to pure hardware is still a HW-SW design. This is the direct result of our technique placing a priority on using software cores. The pure hardware solution requires less area and performs better than our design however, by maintaining a HW-SW design our technique still leaves the flexibility to the designer in the future to add/remove functionality. This is an important attribute of our technique.

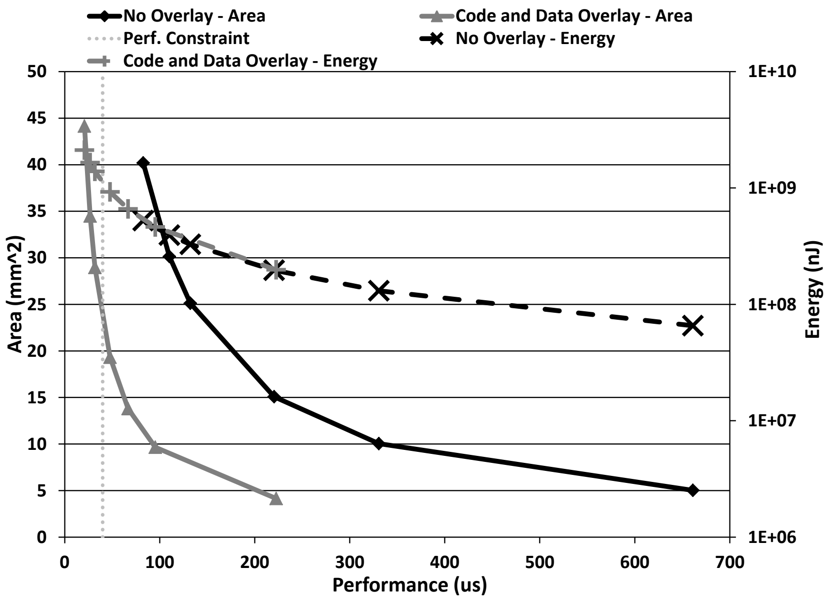


Figure 4.41: fm Area vs. Perf. vs. Energy (sb)

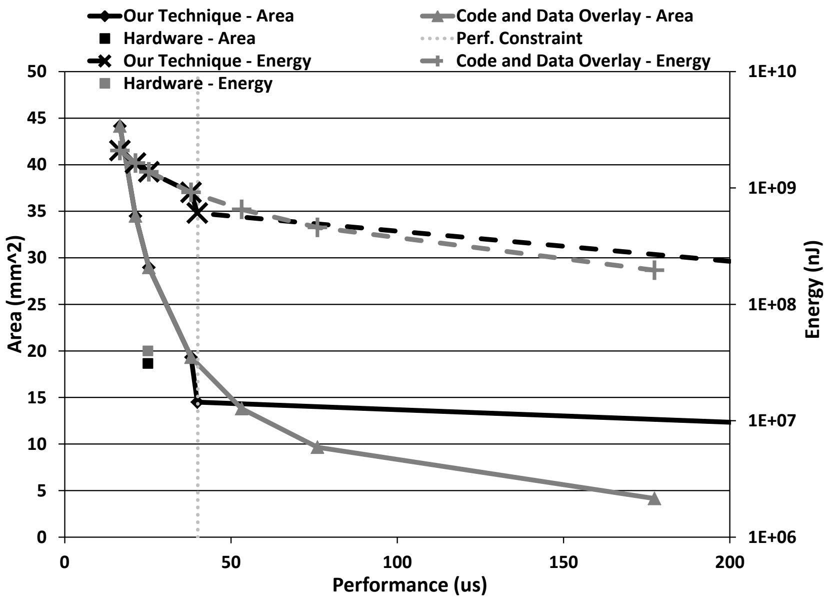


Figure 4.42: fm Area vs. Perf. vs. Energy (ls)

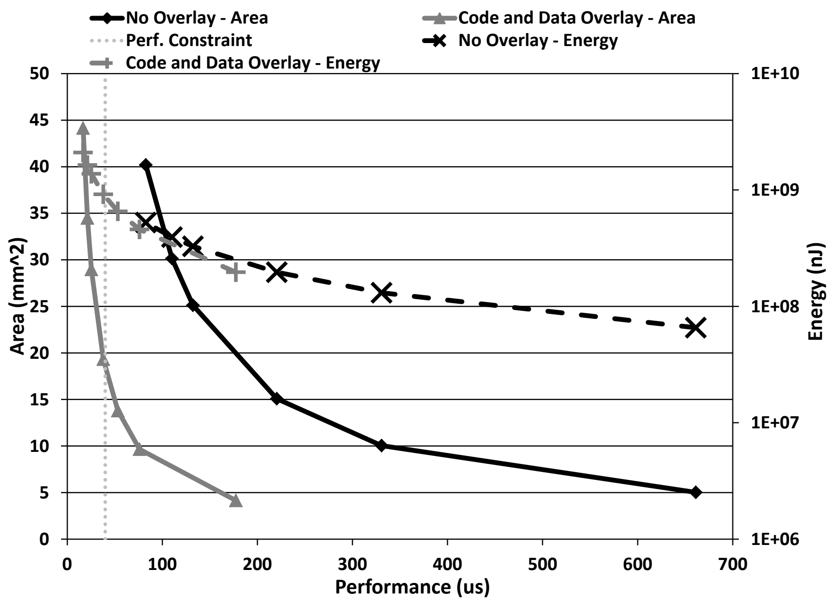


Figure 4.43: fm Area vs. Perf. vs. Energy (ls)

4.7 Summary

We presented a HW-SW co-design synthesis technique for the functional architecture for MPSoC sub-systems. The approach accounts for software and hardware processing elements and generates optimized HW-SW designs that meet given area and performance constraints while giving priority to software cores. Further, the technique generates a memory architecture for the sub-system that accounts for and optimizes for both code and data through the use of overlay schemes. We evaluated our approach through extensive experimentation with streaming application benchmarks through comparisons with various pure software solutions as well as a pure hardware solution. Our technique demonstrated the ability to generate high quality designs that account for the tradeoffs between hardware and software while meeting the area and performance constraints and maintaining a low energy consumption.

Chapter 5

CONCLUSION AND FUTURE WORK

In this dissertation we present novel synthesis techniques for the three stages of the multi-processor System-on-Chip (MPSoC) design process: functional architecture synthesis, memory architecture synthesis, and interconnect architecture synthesis. We begin with presenting a synthesis technique for the interconnect architecture. In particular, we present a Network-on-Chip (NoC) synthesis technique. The technique presented is an extension of the work presented by Srinivasan, et al. [2]. Through modifications to the shortest path graph approach presented in [2] we are able to provide a holistic synthesis technique. The technique provides integrated solutions to mixed communication tyes (transactional/cummulative), port arity, deadlock avoidance, and multiple use-cases. The technique is able to provide these design improvements while using only best effort routers. Experimental results show that our technique is able to generate high quality designs that demonstrate superior performance, area, and power consumption when compared with existing approaches.

In the next phase of our work, we presented a memory architecture synthesis technique. The technique synthesized memory architectures for the sub-systems of a MPSoC. The technique incorporated smart decisions to reduce the memory requirement of code and data for the application filters. This was done through the use of code overlay and data minimization through clique partitioning. Through experimentation we showed the designs generated by our technique were high quality and performed superior when compared to an existing technique in terms of performance, area, and energy consumption.

In the final phase of our work, we presented a functional architecture and memory architecture co-synthesis technique. The technique synthesized the functional architecture for the sub-system of a MPSoC while simultaneously synthesizing the memory architecture. The technique considered the trade-offs of implementing filters in software vs. hardware and made smart decisions when moving filters to hardware accelerators. Further, the technique reduced the memory requirement for code and data through the use of overlay schemes while minimally impacting performance. Through experimentation our technique demonstrated the ability to generate high quality designs in terms of performance, area, and energy consumption when compared with a several pure software solutions and pure hardware solutions.

The dissertation work can be extended in two ways: first, through the addition of a virtual platform to test the designs, and secondly, through the automation of generating the RTL structure of the designs. The discussion of what would be required for these two extensions follows.

5.0.1 Virtual Platform

A virtual platform is a software specification to simulate the functionality of hardware. A virtual platform can behave at a cycle-accurate level, a fast loosely timed level, or somewhere in between. The modeling language SystemC [50] is a natural choice to describe a virtual platform. To build a virtual platform for the MPSoC design flow we would need to model the processing element library, memory component library, and NoC IP blocks library in SystemC. Further, we would modify the stages of the design flow to provide outputs describing the architectures. The virtual platform would be required to read these outputs and generate a software model from the SystemC libraries for the synthesized architectures. The virtual platform would be an useful addition by allowing designers to quickly simulate designs in order to see how they perform prior to implementing them in hardware.

5.0.2 RTL Generation

RTL code describes the functionality of hardware. Using RTL we can perform synthesis to generate actual hardware (ASICs, FPGAS, etc.). The addition of this aspect would hasten the process of taking the designs synthesized by the MPSoC design flow and turning them into a hardware implementation. In order to automate this process we would need to describe the processing element library, memory library, and NoC library in RTL using a hardware modeling language such as VHDL [51]. The stages of the design flow would need to be modified to output characteristics of the designs synthesized. This output would then be read by a netlister which would turn the description of the design into a RTL model. By having the automated generation of RTL models a designer can easily go from design generation synthesis to hardware with little overhead or manual labor. The automatic generation of RTL along with the virtual platform would provide a complete design flow and testing framework for the automated synthesis of highly optimized MPSoC sub-systems.

REFERENCES

[1] L. Benini. Application specific noc design In Proceedings of DATE, 2006.

[2] K. S. Chatha, K. Srinivasan, and G. Konjevod. Automated techniques for synthesis of application-specific network-on-chip architectures IEEE Trans. on CAD on Integrated Circuits and Systems, Vol 27 Issue 8, 2008.

[3] K. Srinivasan, K. S. Chatha, and G. Konjevod. Linear-programming-based techniques for synthesis of network-on-chip architectures IEEE Trans. Very Large Scale Integr. Syst., 2006.

[4] A. Hansson, K. Goossens, and A. Radulescu. A unified approach to constrained mapping and routing on network-on-chip architectures In Proceedings of CODES+ISSS, 2005.

[5] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. D. Micheli. Mapping and configuration methods for multi-use-case networks on chips In Proceeding of ASP-DAC, 2006.

[6] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. D. Micheli. A methodology for mapping multiple use-cases onto networks on chips In Proceedings of DATE, 2006.

[7] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. D. Micheli, and L. Raffo. Designing application-specific networks on chips with floorplan information ICCAD, 2006.

[8] U. Y. Ogras and R. Marculescu. Application-specific network-on-chip architecture customization via long-range link insertion ICCAD, 2005.

[9] W. Dally and B. Towles. Principles and Practices of Interconnection Networks, Morgan Kaufmann, 2004.

[10] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to Algorithms (Second Edition), MIT Press and McGraw-Hill, 2002.

[11] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow In Proceedings of the IEEE, Vol. 75, No. 9, September, 1987.

[12] G. Leary and K.S. Chatha. Holistic Approach to Network-on-Chip Synthesis In Proceedings of CODES+ISSS, 2010.

[13] S. Meftali, F. Gharsalli, F. Rousseau and A. A Jerraya. An Optimal Memory Allocation for Application-Specific Multiprocessor System-on-Chip In Proceedings of International Symposium in System Synthesis (ISSS), 2001.

[14] S. Pasricha and N. Dutt. COSMECA: Application Specific Co-Synthesis of Memory and Communication Architectures for MPSoC In Proceedings of DATE, 2006.

[15] I. Issenin, E. Brockmeyer, B. Durinck and N. Dutt. Data-Reuse-Driven Energy-Aware Cosynthesis of Scratch Pad Memory and Hierarchical Bus-based Communication Architecture for Multiprocessor Streaming Applications IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 8, August 2008.

[16] I. Issenin and N. Dutt. Data Reuse Driven Memory and Network-on-Chip Co-Synthesis IFIP Embedded System Design: Topics, Techniques and Trends, 2007.

[17] M. Manchiero, G. Palermo, C. Silvano and O. Villa. Exploration of Distributed Shared Memory Architectures for NoC-based Multiprocessors Journal of Systems Architecture: the EUROMICRO Journal Volume 53 Issue 10, October, 2007

[18] S. Pandey and R. Drechsler. Slack Allocation Based Co-synthesis and Optimization of Bus and Memory Architectures for MPSoCs In Proceedings of DATE, 2008.

[19] C. Lee, S. Kim and S. Ha. A Systematic Design Space Exploration of MPSoC Based on Synchronous Data Flow Specification Journal of Signal Processing Systems, Vol 58, 2010.

[20] A. Jantsch. Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation, Morgan Kaufmann Publishers, 2004.

[21] W. Thies, M. Karczmarek and S. Amarasinghe. Streamit: A language for streaming applications In Proceedings of International Conference on Compiler Construction, 2002.

[22] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow In Proceedings of the IEEE, Vol. 75, No. 9, September, 1987.

[23] G. Leary and K.S. Chatha. Holistic Approach to Network-on-Chip Synthesis In Proceedings of CODES+ISSS, 2010.

[24] G. Leary, W. Che, and K.S. Chatha. System-level Synthesis of Memory Architecture for Stream Processing Sub-Systems of a MPSoC In Proceeding of DAC, 2012.

[25] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor SIGPLAN, Volume 31, Issue 9, 1996.

[26] W.H. Wolf. Hardware-software codesign of embedded systems In Proceedings of IEEE, Volume 82, 1994.

[27] R.K. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems IEEE Design Test Computation Magizine, 1994.

[28] W. Che, A. Panda, and K.S. Chatha. Compilation of Stream Programs for Multicore Processors that Incorporate Scratchpad Memories In Proceedings for DATE, 2010.

[29] W. Che and K. S. Chatha. Scheduling of Stream Programs onto SPM Enhanced Processors with Code Overlay In Proceedings ESTIMEDIA, 2011.

[30] W. Che and K. S. Chatha. Scheduling of Synchronous Data Flow Models on Scratchpad Memory Based Embedded Processors In Proceedings ICCAD, 2010.

[31] T.L. Adam, K.M. Chandy, and J. R. Dickson. A Comparison of List Schedules for Parallel Processing Systems Comm. ACM, Volume 17, Issue 12, 1974,

[32] E.G. Coffman, Jr., and P.J. Denning. Operating Systems Theory. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[33] E. B. Fernandez and B. Bussell. Bounds on the number of processors and time for multiprocessor optimal schedules IEEE Trans. Comput. Volume C-22, Issue 8, 1973.

[34] M. R. Garey, R.L. Graham, and D. S. Johnson. Performance Guarantees for Scheduling Algorithms Oper. Res., Volume 26, Issue 1, 1978.

[35] T. C. Hu. Parallel Sequencing and Assembly Line Problems Oper. Res., Volume 9, 1961.

[36] H. Kasahara and S. Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing IEEE Trans. Comput., Volume C-33, Issue 11, 1984.

[37] M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. New York: Freeman, 1979.

[38] S. Prakash and A. Parker. SOS: Synthesis of application-specific heterogeneous multiprocessor systems Journal on Parallel Distributed Computing, Volume 16, 1992.

[39] M. Schwiegershausen and P. Pirsch. Formal approach for the optimization of heterogeneous multiprocessors for complex image processing schemes In Proceedings DATE, 1995.

[40] J. D. Ambrosio and X. Hu. Configuration-level hardware-software partitioning for real-time systems In Proceedings International Workshop Hardware-Software Codesign, Volume 14, 1994.

[41] R.P. Dick and N.K. Jha. Mogac: a multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems In Proceedings for ICCAD, 1997.

[42] Y.J. Chen, C.L. Yang, and Y.S. Chang. An architectural co-synthesis algorithm for energy-aware network-on-chip design JSA, Volume 55, Issue 5-6, 2009.

[43] Y.J. Chen, C.L. Yang, and P.H. Wang. PM-COSYN: PE and Memory Co-Synthesis for MPSoCs In Proceedings for DATE, 2010.

[44] W. Thies, M. Karczmarek and S. Amarasinghe. Streamit: A language for streaming applications In Proceedings of International Conference on Compiler Construction, 2002.

[45] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory : A Design Alternative for Cache On-chip memory in Embedded Systems IN Proceedings of CODES, 2002.

[46] Forte. Forte Cynthesizer http://www.forteds.com/products/cynthesizer.asp

[47] Synopsys. Synopsys PrimePower http://www.synopsys.com/home.aspx

[48] ARM. ARM Cortex R4 Series of Cores. http://www.arm.com/products/processors/cortex-r/cortex-r4.php

[49] A. Jantsch. Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation, Morgan Kaufmann Publishers, 2004.

[50] Accellera. The SystemC Standard. http://www.accellera.org/downloads/standards/systemc

[51] IEEE. VHDL Standard. http://ieeexplore.ieee.org/xpl/

[52] LP_solve. LP_solve Open Source Project. http://sourceforge.net/projects/lpsolve/

[53] University of Michigan. Parquet Floorplanner. http://vlsicad.eecs.umich.edu/BK/parquet/

APPENDIX A

NETWORK-ON-CHIP SYNTHESIS

In this appendix we will discuss the framework for the Network-on-Chip (NoC) synthesis technique. In the following, we will first discuss the file structure of the code base for the technique. Next, we will discuss the environment setup followed by the execution command. We will then discuss the file format of the inputs and outputs.

File Structure

The code base for the technique is written exclusively in the C++ standard library. The files for the technique include:

- Makefile: The Makefile contains the commands to build the synthesis technique.
- global.h: This file includes global definitions. Things like TRUE, FALSE, etc. These are definitions that will be used throughout the entire code base.
- objects.h/objects.cpp: This file contains the definitions for all of the objects in the code along with the function definitions to accessing and manipulating the objects. Some of the objects would include: use cases, transactions, shortest path graph, etc.
- main.h/main.cpp: This file has the main routine within it. The main routine controls the execution flow of the program.
- read_files.h/read_files.cpp: This file reads in the input files and creates the objects accordingly.

- router_placer.h/router_placer.cpp: This file allocates routers at the appropriate locations on the floorplan.

- sa_core_to_router_mapping.h/sa_core_to_router_mapping.cpp: This file performs the core to router mapping. This file uses a simulated annealing approach to find a mapping in which cores are mapped to a router within a certain maximum distance.

- opt_core_to_router_mapping.h/opt_core_to_router_mapping.cpp: This file performs the core to router mapping. This file uses a optimized core to router mapping technique presented by Srini et al. [2]. The approach maps a core to one of the routers located at its four corners. The approach finds a mapping with minimal power consumption.

- shortest_paths_generator.h/shortest_paths_generator.cpp: This file contains the routines to generate the shortest path graphs for each of the communication traces in the application.

- communication_resolution.h/communication_resolution.cpp: This file contains the routines to perform the commuinication resolution stage of the algorithm. This stage ensures that communication does not interfere with each by adding additional ports to router when necessary.

- deadlock_avoidance.h/deadlock_avoidance.cpp: This file contains the routines to eliminate potential deadlocks from the shortest paths. The file will convert the global shortest path graph into the global channel dependency graph. From there it will locate cycles and break the cycles before reflecting the changes back onto the shortest paths of the traces.

- port_arity_resolution.h/port_arity_resolution.cpp: This file contains the routines to perform the port arity resolution. This stage will find routers that violate the maximum port arity of the router by adding additional routers and rerouting traces.

- lp.h/lp.cpp: This file contains the routines to generate and solve the LP formulation. The LP formulation is described in Srini et al. [2].

- print_output_files.h/print_output_files.cpp: This file prints the output files containing all of the information about the NoC design.

- push_relabel.h/push_relabel.cpp: This is a helper function used to find the shortest paths.

Environment Setup

In order to execute the NoC synthesis technique the system environment must be properly setup. First, C++ compiler must be installed. The code has been tested and proven to function properly with *gcc* version 4.7.2. The environment will also need LP_solve [52] installed. LP_solve will need the additional package *xli_XPRESS* installed. The code has been tested using LP_solve version 5.5. The following environment variables must be updated:

- PATH=$PATH:<lp_solve_dir>/lp_solve_5.5/lp_solve/bin/ux32

- LD_LIBRARY_PATH=<lp_solve_dir>/lp_solve_5.5/xli/xli_XPRESS/bin/ux32

Execution Command

The execution of the program is performed from the directory containing the input files. The command is:

$holistic < problem\_name >$

where problem name is the name of the benchmark you want a NoC synthesized for. All input files must begin with the problem name. For example, $< problem\_name > .floorplan$.

Input Files

The tool requires several input files to describe the problem instance. The required input files are as follows:

- <problem_name>.floorplan: This file is a tab delimited file containing the placement, size, and characteristics of each core in the floorplan. Each core is described on a single line in the following manner:

  FILE FORMAT:

  ID X_DIM Y_DIM X_MIN Y_MIN X_MAX Y_MAX FREQ. MLL D_WIDTH

  where id denotes the id of the core, x_dim and y_dim describe the x and y dimensions of the core, x_min, x_max, y_min, and y_max describe the coordinates of the location for the core in the floorplan. $freq.$ is the operating frequency of the core, MLL is the distance a packet can travel to/from the core before being buffered, and d_width is the width of the ports to the core.

- <problem_name>.applications: This file is a tab delimited file containing the information for each trace in the application(s). The file describes each individual application first followed by the use cases.

  FILE FORMAT:

  Application ID

  ID SOURCE SINK BW PERIOD #_TRANS TYPE START1 START2

  Use_case ID PERCENT APP_IDs

  where *id* after application denotes the id for the application. ID denotes the id for the trace, $SOURCE$ and $SINK$ denote the id of the core for the source and sink for the trace, BW and $PERIOD$ denote the bandwidth of the trace and the execution period for the trace. $\#\_TRANS$ is the number of transactions in the trace (if any). It will then be followed by a list of three values: $TYPE$, $START1$, and $START2$ which describe the type and start window for each transaction. The use cases are described with an ID, denoting the use case id, $PERCENT$ denoting how often the use case is active, and a list of APP_IDs which denote which applications are included in the use case.

- <problem_name>.labeling: This file is a tab delimited file containing a mapping of the actual name of the cores to the id used in the rest of the files. The file format is as follows:

  FILE FORMAT:

  ID $NAME$

  where ID denotes the id, and $NAME$ denotes the common name of the core.

- <problem_name.router_types: This file is a tab delimited file containing the descriptions of the available routers in the router library. The file format is as follows:

  FILE FORMAT: ID M_PORTS M_CORES P_WIDTH FREQ R_ENERGY L_ENERGY

  where ID is the id for the router element, M_PORTS is the maximum number of ports the router supports, M_CORES is the maximum number of cores that can be conencted to the router, P_WIDTH is the width of the ports of the router, $FREQ$ is the operating frequency of the router, R_ENERGY is the energy consumption of the router, and L_ENERGY is the energy consumption of the router links.

Output Files

In this section we were discuss the output files of the synthsis technique. The files are as follows:

- floorplan.pdf: This file provides a visualization of the base floorplan.
- router_mapping.pdf: This file provides a visualization of the core-to-router mapping on the floorplan.
- topology.pdf: This file provides a visualization of the generated NoC topology (graph structure).
- device_descriptions.txt: This file provides a detailed description of each of the routers included in the file topology, including the number of input/output ports, the frequency, etc.

- connectivity.txt: This file describes how the routers are connected together.

APPENDIX B

MEMORY ARCHITECTURE SYNTHESIS

In this appendix we will discuss the framework for the memory synthesis technique. In the following, we will first discuss the file structure of the code base for the technique. Next, we will discuss the environment setup followed by the execution command. We will then discuss the file format of the inputs and outputs.

File Structure

The code base for the technique is written exclusively in the C++ standard library. The files for the technique include:

- global.h: This file contains variable definitions that are used by every file.
- objects.h/objects.cpp: This file contains all the information for the objects in the code. The files also contain the functions necessary to access and manipulate the objects.
- main.h/main.cpp: This file contains the main execution flow of the program.
- read_files.h/read_files.cpp: This file contains the necessary functions to read the input files. This file also contains a function to generate the necessary output files.
- pass.h/pass.cpp: This file generates the PASS execution schedule for the SDF input specification.
- floorplan.h/floorplan.cpp: This file performs floorplanning on the memory and cores.

- lifetimes.h/lifetimes.cpp: This file calculates the lifetimes of the data blocks and the code segments.
- gen_traces.h/gen_traces.cpp: This file generates the communication traces necessary for the cores/memories to communicate.
- memory.h/memory.cpp: This file generates and optimizes the memories.
- data_memory.h/data_memory.cpp: This file performs the optimizations on the data blocks in memory. In particular, it performs the clique partitioning and sharing of memory regions between non-interfering data blocks.
- clique_partitioning.h/clique_partitioning.cpp: This file performs the clique partitioning algorithm.
- code_memory.h/code_memory.cpp: This file contains the functions to generate code overlays.
- Makefile: This file contains the commands to build the program.

Environment Setup

In order to execute the mmory synthesis technique the system environment must be properly setup. First, C++ compiler must be installed. The code has been tested and proven to function properly with *gcc* version 4.7.2. The environment will also need the Parquet Floorplanner [53]. The following environment variables must be updated:

- PATH=$PATH:<parquet_dir>/bin

Execution Command

The execution of the program is performed from the directory containing the input files. The command is:

$mem\_synth < problem\_name >$

where problem name is the name of the benchmark you want a memory synthesized for. All input files must begin with the problem name. For example, $< problem\_name > .filters$.

Input Files

In this section we will discuss the input files to the synthesis tool. The tool requiers several input files as described:

- <problem_name>.filters: This file contains the relevant information about the filters of the SDF. The file format is as follows:

  FILE FORMAT:

  ID NAME C_SIZE EXEC

  where ID is the id number for the filter, $NAME$ is the common name for the filter, C_SIZE is the size of the code for the filter, and $EXEC$ is the amount of time to execute the filter on the core it is mapped.

- <problem_name>.cores: This file contains the characteristics of the cores in the problem. The file format is as follows:

FILE FORMAT:

ID X_DIM Y_DIM

where ID is the id of the core, X_DIM is the x dimension of the core, and Y_DIM is the y dimension of the core.

- <problem_name>.mapping: This file provides the mapping of the filters to the cores. The file format is as follows:

  FILE FORMAT:

  FILTER CORE

  where $FILTER$ is the id of the filter being mapped to the core and $CORE$ is the id of the core the filter is mapped to.

- <problem_name>.sdf_desc: This file describes the SDF specification for the application. The SDF is described in a graph format. The file format is as follows:

  FILE FORMAT:

  SOURCE SINK D_GEN D_CON

  where $SOURCE$ is the id of the source filter, $SINK$ is the id of the sink filter, D_GEN is the amount of data generated by the source filter, and D_CON is the amount of data being consumed by the sink filter.

- <problem_name>.memory_types: This file defines the various memory types available to the synthesis technique. The file format is as follows:

  FILE FORMAT:

  ID N_PORTS SIZE X_DIM Y_DIM POWER L_POWER LAT LAT2 D_WIDTH LINK where ID is te id of the memory type, N_PORTS is the number of ports on the memory, $SIZE$ is the capacity of the memory, X_DIM isthe x dimension of the memory, Y_DIM is the y dimension of the memory, $POWER$ is the power consumption of the memory, L_POWER is the power consumption of the memory links, LAT is the access latency for the first byte of data, $LAT2$ is the latency for successive data accesses, D_WIDTH is the width of the data ports, and $LINK$ is the maximum link length to/from the memory.

Output Files

In this section we discuss the files produced as output to the memory synthesis technique. The output files are as follows:

- <problem_name>.applications: This file describes the communication between the cores and memories. This file is an input to the NoC synthesis technique.
- <problem_name>.floorplan: This file describes the floorplan of the cores and memories. This file is also an input to the NoC synthesis technique.
- <problem_name>.labeling: This file provides a mapping of the ids of the cores and memories to their common name. This file is an input to the NoC synthesis technique.
- memory_description.txt: This file describes the memories in the solution. The file provides information relating the the size, number of ports, what data is stored, etc.

APPENDIX C

FUNCTIONAL ARCHITECTURE SYNTHESIS

In this appendix we will discuss the framework for the functional synthesis technique. In the following, we will first discuss the file structure of the code base for the technique. Next, we will discuss the environment setup followed by the execution command. We will then discuss the file format of the inputs and outputs.

File Structure

The code base for the technique is written exclusively in the C++ standard library. The files for the technique include:

- global.h: This file contains variable definitions that are used by throughout the code base, such as TRUE, FALSE, etc.
- objects.h/objects.cpp: This file contains the descriptions of all of the objects used in the code base. The files also contain the functions necessary to access and manipulate the objects.
- main.h/main.cpp: This file provides the overall execution flow of the program.
- read_files.h/read_files.cpp: This file provides functions to read the input files describing the problem intance.
- pass.h/pass.cpp: This file generates the execution scehdule for the software cores. It includes functions to generate both a minimum buffer schedule and a least switching schedule.

- initial_solution.h/initial_solution.cpp: This file provides the functions necessary to generate the initial solution. This includes the optimization of the memory through the use of data and code overlays.
- hw_replication.h/hw_replication.cpp: This file provides a function to replicate the hardware accelerators in order to expand the hardware accelerator library.
- optimize_design.h/optimize_design.cpp: This file provides functions to optimize the design. This optimization is the iterative process of moving filters out of software and onto hardware accelerators.
- hw_solution.h/hw_solutions.cpp: Thsi file provides the routine to generate a pure hardware solution. This is called when a HW-SW solution can not be found.
- optimize_energy.h/optimize_energy.cpp: This file contains functions to attempt to minimize the energy consumption.
- output_design.h/output_design.cpp: This file contains functions to generate the output files.
- Makefile: This file contains the methods to build the program.

Environment Setup

In order to execute the functional synthesis technique the system environment must be properly setup. First, C++ compiler must be installed. The code has been tested and proven to function properly with gcc version 4.7.2.

Execution Command

The execution of the program is performed from the directory containing the input files. The command is:

$func\_synth < problem\_name >$

where problem name is the name of the benchmark you want a functional architecture synthesized for. All input files must begin with the problem name. For example, $< problem\_name >$ ..filters

Input Files

In this section we will discuss the input files to the synthesis tool. The tool requires several input files as described:

- <problem_name>.filters: This file contains the description of the filters in the problem. The file format is as follows:

  File Format:

  ID NAME SIZE

  where ID is the id of the filter, $NAME$ is the common name for the filter, and $SIZE$ is the size of the filter in bytes.

- <problem_name>.memory_library: This file describes the characteristics of the memory elements. The file format is as follows:

  File Format:

  ID NAME X_DIM Y_DIM WIDTH FREQ LAT_1 LAT_2 ENERGY SIZE MLL where ID is the id of the memory type, $NAME$ is the common name for the memory type, X_DIM is the x dimension, Y_DIM is the y dimension, $WIDTH$ is the data width of the ports, $FREQ$ is the operating frequency of the memory, LAT_1 is the latency to access the first byte of data, LAT_2 is the latency to access successive bytes of data, $ENERGY$ is the energy consumption per access, $SIZE$ is the size of the memory in bytes, and MLL is the maximum link length that can be attached to the memory.

- <problem_name>.pe_library: This file describes the characeristics of the processing element libray (both hardware and software processing elements). The file format is as follows:

  File Format:

  ID NAME X_DIM Y_DIM WIDTH FREQ/EXEC ENERGY MLL TYPE

  where ID is the id of the processing element, $NAME$ is a common name for the processin element, X_DIM is the x dimension of the processing element, Y_DIM is the y dimension of the processing element, $WIDTH$ is the data width of the processing element, $FREQ/EXEC$ is either the operating frequency (software) or the execution time of the hardware accelerator, $ENERGY$ is the energy consumption of the processing element, MLL is the maximum link length for the processing element, and $TYPE$ is the type of the processing element (HW or SW).

- <problem_name>.sdf: This file describes the SDF specification for the application. The file format is as follows:

File Format:

SOURCE SINK D_GEN D_CON

where $SOURCE$ is the id of the source filter, $SINK$ is the id of the sink filter, D_GEN is the amount of data being generated by the source filter, and D_CON is the amount of data being consumed by the sink filter.

Output Files

In this section we will discuss the output files of the functional synthesis technique. The output files are as follows:

- core_description.txt: This file describes the characteristics of the cores in the solution.
- memory_description.txt: This file describes the characteristics of the memories in the solution. This file includes the number of memory regions and what is mapped to each region (if applicable).
- connectivity.txt: This file describes which memories are attached to which cores.
- mapping.txt: This file provides a mapping of the filters of the SDF to the cores.
- communication.txt: This file provides the communication between the cores and memories.

Due to the functional synthesis technqiue utilizing estimates to determine the performance of the design. The output files need to be postprocessed to generate the proper input to the NoC synthesis technique. The applications input file needs to be generated along with a proper floorplan. These steps were purposefully omitted from the synthesis techniqe due to the large overhead required to constantly update the floorplan and communication traces.