

**SYNTHESIZING APPLICATION
SPECIFIC MULTIPROCESSOR
ARCHITECTURES FOR PROCESS
NETWORKS**

BASANT KUMAR DWIVEDI



**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI**

©Indian Institute of Technology Delhi - 2005

All rights reserved.

**SYNTHESIZING APPLICATION
SPECIFIC MULTIPROCESSOR
ARCHITECTURES FOR PROCESS
NETWORKS**

by

BASANT KUMAR DWIVEDI

Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of Doctor of Philosophy

to the



Indian Institute of Technology Delhi

July 2005

Certificate

This is to certify that the thesis titled **Synthesizing Application Specific Multi-processor Architectures for Process Networks** being submitted by **Basant Kumar Dwivedi** for the award of **Doctor of Philosophy in Computer Science & Engg.** is a record of bona fide work carried out by him under our guidance and supervision at the **Dept. of Computer Science & Engg., Indian Institute of Technology Delhi**. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Anshul Kumar

Professor

Dept. of Computer Science & Engg.

Indian Institute of Technology Delhi

M. Balakrishnan

Professor

Dept. of Computer Science & Engg.

Indian Institute of Technology Delhi

Acknowledgments

I am greatly indebted to my supervisors Prof. Anshul Kumar and Prof. M. Balakrishnan for their invaluable technical guidance and moral support. I would like to thank Anup Gangwar for his feedbacks during our informal discussions and my other colleagues specially Manoj Jain, Subhajit Sanyal and Parag Chaudhuri for their cooperation and support. I would also like to thank the staff of Philips, FPGA and Vision laboratories, IIT Delhi for their help. I am very thankful to Naval Research Board, Govt. of India for being enabler of my work.

My father Hriday Narayan Dwivedi and mother Anupma Dwivedi showed immense patience and provided me great moral support during the course of my work. My other family members specially my wife Swati and elder brother Hemant also helped me a lot in getting me this far.

Abstract

Today Embedded Systems on a Chip (SoCs) can be found in a large variety of applications like image processing, computer vision, networking, wireless communication etc. Typically these applications demand high throughput and impose real time processing constraints. Many of these applications, specially in the domain of image processing and computer vision, also offer significant amount of functional and data parallelism. Multiprocessors, optimized for these applications have become one of the obvious choices by exploiting the parallelism effectively.

There are two approaches which are followed for the synthesis of application specific multiprocessor architectures. Starting point in the first approach is directed acyclic graph (DAG) derived from the given application(s). In this approach, static scheduling is performed to solve the synthesis problem which essentially consists of architectural resource allocation, binding of application components to architecture and scheduling. This approach mainly suffers from the limitations such as loops are represented as single node of the task graph derived from the application, effect of local communication is not properly modeled and static scheduling is performed using worst case behavior making the system over-designed.

Another alternative is to start from the application specification where application is modeled as process networks. Here, processes communicate through FIFO queues. Unlike task graphs, in process networks, computation and communication are interleaved and parallelism present in the application is made explicit.

In the past, researchers have addressed the problem of automatic synthesis of optimal architecture for process networks only partially. In this thesis, we address the automatic synthesis of optimal architectures for process networks in an integrated manner by developing a framework for this problem. In this framework, architectural resource allocation

is performed in which resources (processors, memories etc.) are chosen from a component library. Alongwith resource allocation, we perform binding of various processes of the process network onto processors and queues onto memories. Here we assume that a dynamic scheduler will take care of run-time scheduling requirements. Another novel feature of our work is synthesis of interconnection network (IN) by exploiting communication patterns present within the application process network. As part of IN synthesis, we have also developed new analytical models to estimate the queuing delays in heterogeneous multiprocessors with non-uniform memory accesses. While performing validation, we found that there are no benchmarks available with applications modeled as process network. Further, developing such benchmarks for real applications is very time consuming process and not easily portable across different PN frameworks. Hence, we developed a new random process network generator (RPNG) which can create sample process networks with defined characteristics quickly and aid in quick validation and comparison of Multiprocessor Systems-on-Chip (MpSoC) synthesis techniques.

Our experiments on real-life application MPEG-2 decoder and a number of process networks generated by RPNG, show that the our synthesis framework is quite scalable and produces solutions of reasonable quality quickly. These experiments also prove the usefulness of RPNG as an enabler for MpSoC synthesis research.

Contents

List of Figures	v
List of Tables	ix
1 Introduction	1
1.1 Need for Multiprocessor Systems-on-Chip (MpSoC)	1
1.2 Role of Architecture Customization	2
1.3 Objectives	3
1.4 Architecture Design Space	4
1.4.1 Computation Units	5
1.4.2 Interconnection Network	5
1.4.3 Memory Architecture	6
1.5 Application Specific MpSoC Design Subproblems	7
1.6 Previous Work	8
1.6.1 Simulation Based Approaches	9
1.6.2 Analytical and Scheduling Based Approaches	10
1.6.3 Hardware Software Co-design Infrastructures	12
1.6.4 MpSoC Synthesis for Process Networks	13
1.7 Our Approach	15

1.8	Thesis Outline	15
2	Overall Methodology	17
2.1	System Model	17
2.1.1	Models of Computation	17
2.1.2	Application Model	22
2.1.3	Architectural Component Library	23
2.2	Hardware Estimation	24
2.2.1	Hardware Estimator Tool	24
2.2.2	Validation of Hardware Estimator Tool	26
2.3	Software Estimation	26
2.4	Multiprocessor Architecture Synthesis	29
2.5	Summary	30
3	Estimation of Queuing Delays for Heterogeneous Multiprocessors	33
3.1	Related Work	34
3.2	System Architecture	35
3.3	Analysis of Burst Mode Traffic	37
3.4	Analysis of Non-Burst Mode Traffic	39
3.4.1	Single memory case	40
3.4.2	Multiple memory case	45
3.5	Experiments and Results	46
3.6	Summary	51
4	MpSoC Synthesis Problem Formulation	53
4.1	Application Parameters	54
4.2	Architectural Parameters	56
4.3	MILP for Computing Resources and Application Mapping	58

4.3.1	Decision Variables	58
4.3.2	Mapping Constraints	59
4.3.3	Performance Constraints	60
4.4	MILP for Communication Architecture	64
4.5	Summary	66
5	MpSoC Synthesis Heuristics	67
5.1	Construction of Initial Solution	68
5.1.1	Definition of Partial Map	70
5.1.2	Creating Adjacent Process List	70
5.1.3	Computation of Partial Map	72
5.1.4	Merging of shared memories	75
5.1.5	Algorithm Complexity	76
5.2	Iterative Refinements of Initial Solution	77
5.3	Power Optimization	80
5.4	Correctness of Synthesized Architecture	83
5.5	Summary	84
6	Random Process Network Generator (RPNG)	85
6.1	Introduction	85
6.2	Motivation for RPNG	87
6.3	Structure of a Process Network	90
6.4	Process Network Generation	92
6.4.1	Initialization of Process Network	94
6.4.2	Addition of a New Process	95
6.4.3	Addition of a New Channel	96
6.5	Code Generation	98
6.5.1	Process Flow Level and Condition for Deadlock Free Execution	98

6.5.2	Insertion of Statements in Processes	101
6.6	Creation of Database	104
6.7	Summary	107
7	Experimental Results	109
7.1	MPEG-2 Video Decoder Case Study	109
7.1.1	Using MILP Solver	109
7.1.2	Using Heuristic Based Framework	112
7.2	Experiments Using RPNG	113
7.3	Validation	119
7.4	Summary	120
8	Conclusions and Future Work	121
8.1	Contributions	121
8.2	Future Work	122
	Bibliography	127

List of Figures

1.1	Customization Opportunities in Multiprocessors	4
1.2	Typical MpSoC architecture	7
2.1	Srijan synthesis framework	18
2.2	Example process network	22
2.3	Sample MpSoC architecture	23
2.4	<i>HwEst</i> : Hardware Estimator Tool	25
2.5	Software Estimation Methodology	28
2.6	Example illustrating MpSoC synthesis for application modeled as process network	30
3.1	Architectures considered	35
3.2	Mutual interference of read/write requests on queues Q_1 and Q_2	37
3.3	Two state request resubmission model	40
3.4	States of a processor	41
3.5	States of a processor making request to multiple memories	46
3.6	MRR vs. Delta for single memory	49
3.7	MRR vs. Avg. IRR for single memory	50
3.8	MRR vs. Delta for multiple memory	51
3.9	RMS error vs. Delta	52

4.1	Synthesis example	54
5.1	Annotated MPEG-2 video decoder process network	71
5.2	Example partial map	71
5.3	CRG of MPEG-2 video decoder	72
5.4	Example solution tree	73
5.5	Iterative Refinement Tree	78
5.6	Voltage scaling in a processor	81
6.1	MPEG-2 video decoder	86
6.2	Unrolled MPEG-2 video decoder	88
6.3	Computation within MPEG-2 video decoder	89
6.4	Example process network	91
6.5	Structure of a process	92
6.6	Various stages of process network graph generation	95
6.7	Nesting and flow levels in example process network	99
6.8	WRITE statements in process PL0	102
6.9	Statements in process PL0	103
6.10	Statements in process IntP_0	104
6.11	Sample attributes for database	106
6.12	Partial database	107
7.1	MPEG-2 Video Decoder Process Network	110
7.2	Annotated MPEG-2 video decoder process network	110
7.3	Synthesized architecture and mapping for Table 7.1	113
7.4	Synthesized architecture and mapping for Table 7.2	114
7.5	Attributes to generate database by RPNG	116
7.6	Process Network generated by RPNG using parameters of Figure 7.5	117

7.7	Partial database generated by RPNG for attributes of Figure 7.5	118
7.8	Architecture synthesized for process network of Figure 7.6	119

List of Tables

2.1	validation of upperbound estimates	27
4.1	Notations for application parameters	55
4.2	Notations for architecture parameters	57
5.1	Partial map matrix	76
5.2	Energy consumed by processes on processor PR_k at different clock periods	83
5.3	Example illustrating steps of Algorithm 3	83
6.1	Parameters for process network graph generation	94
6.2	Fanout probabilities for Figure 6.6(d)	97
6.3	Fanin probabilities for Figure 6.6(f)	97
7.1	Inputs for MILP based solver	112
7.2	Process and processor parameters	114
7.3	Runtime taken by MILP solver to solve various problem instances	117
7.4	Experiments to test scalability of MpSoC synthesis framework	119
7.5	Quantitative comparison of synthesis problem solved by MILP solver vs. Heuristic based implementation	120

Chapter 1

Introduction

1.1 Need for Multiprocessor Systems-on-Chip (MpSoC)

Today Embedded Systems on a Chip (SoCs) can be found in a large variety of applications like image processing, computer vision, networking, wireless communication etc. Typically these applications demand real time processing and high throughput. Many of these applications, specially in the domain of image processing and computer vision, also offer significant amount of functional and data parallelism. Multiprocessors, fine tuned for these applications have become one of the obvious choices because of the computing power they offer by exploiting the parallelism effectively.

Earlier, most of the embedded systems did not use multiprocessors. They essentially comprised of a processor and some hardware built around it. The software was used for achieving fast turn-around time, while the hardware was used to speed up critical portions of the system. On the other hand, shared memory multiprocessors are very popular in the server segment. Earlier multiprocessor architectures were not used in embedded systems due to large latency of off-chip processor communication and high cost. Now as the

technology is slowly moving towards nanometer era, it has become possible to fabricate billions of transistors on a single chip. This also dramatically reduces the communication cost among processors in a multiprocessor architecture when implemented on a single chip. Technology being the enabling factor, these days vendors like Cradle Technologies [1], Texas Instruments [2], XILINX [3] etc. have already started offering MpSoCs.

The main motivating factor behind using MpSoC, is the technology trends and great amount of parallelism present in several applications which demand real time processing. Another aspect which motivates the need for multiprocessor SoCs is control. While in a single chip SoC with hardware accelerators, the single processor is expected to coordinate the various accelerators, in a MpSoC this task can be offloaded to one processor leaving others free to participate in computation. In fact, MpSoCs with multiple processors, where one is dedicated for control and others are suitable for digital signal processing have already started appearing in many products including multimedia phones [2].

1.2 Role of Architecture Customization

Architecture customization leads to design solutions which are cheaper cost-wise as well as satisfy the constraints on performance and power consumption tightly. The *General Purpose Computing* domain doesn't offer much opportunity for architecture customization as it is not known in advance which application will run on the target architecture. However, in embedded systems the application is known in advance and as a consequence it is possible to analyze the application rigorously and fine tune the architecture. Thus Application Specific Instruction Processors (ASIPs) are important components of SoCs.

As an example, consider a image processing application, where the image itself is represented as an array of bytes. This offers a possible customization opportunity wherein the datapath is itself designed keeping the above fact in mind. This decreases the word size of the ALU, bus, registers etc. all leading to a design solution which is much cheaper. Another

example of customization opportunity is the memory access pattern of some applications. If, say the application accesses the memory elements in FIFO order, then the memory architecture itself can be fine tuned alongwith the address generation unit.

At the system level, the inter-component communications network can be customized, specific tasks can be offloaded to dedicated processors, the number and types of processors can be appropriately chosen etc. We present a detailed set of architecture components/parameters which can be fine tuned, both at the system as well as the subsystem level in Section 1.4.

1.3 Objectives

This thesis is a part of the research project **Srijan** [5] in which a design methodology for ASIP based multiprocessor architectures is being developed. Out of the overall design space, in this thesis, we focus on multiprocessor system architecture with the following objectives.

- To formulate the problem of synthesis of SoC multiprocessor for process networks.
- To develop a framework in which the above synthesis problem can be solved for cost as well as power optimization.
- To develop estimation models required for the above synthesis problem so that queuing delays for heterogeneous multiprocessors with non-uniform memory access for burst as well as non-burst mode traffic can be estimated.
- To work out a framework which allows one to perform large number of architecture synthesis experiments.

1.4 Architecture Design Space

Figure 1.1 shows the essential components of an application specific multiprocessor system. A number of processors and co-processors access memory and communicate with each other using an interconnection network. Broadly these architectures could be distributed memory or shared memory. In the distributed memory case, processors do not share the memory space and communication is through message transfers. In the shared memory case all the processors share the address space. For single chip multiprocessors, we intend to explore shared memory architectures.

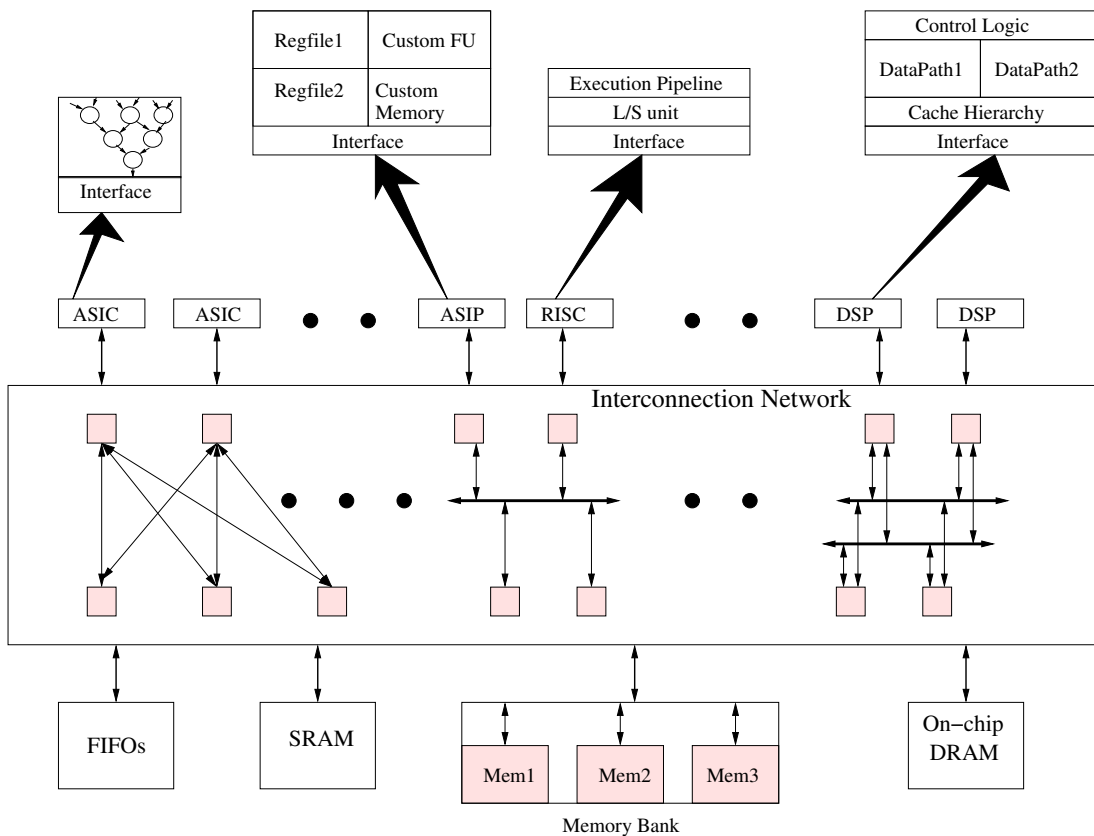


Figure 1.1: Customization Opportunities in Multiprocessors

1.4.1 Computation Units

The overall execution time of the application mapped onto an architecture depends on a number of factors such as memory hierarchy, communication structure etc, however type of the processor remains a key contributor. For example, any signal processing application will run faster on a DSP, whereas any control dominated application will not be able to exploit the resources of such processors effectively. Performance can be enhanced if different parts of the application run on different processors which are optimized for those characteristics. For example, an application which consists of a mix of control specific and digital signal processing will execute faster if former is mapped onto a RISC and later is mapped onto a DSP. Now, it is clear that performance requirements of an application, which has enough parallelism and has different kind of processing requirements, will lead to a heterogeneous multiprocessor configuration i.e. with different types of processors.

Some of the significant variations in processors are due to the following features.

- **Processor Type:** ASICs, VLIW or RISC ASIPs, DSPs.
- Instruction set architecture of the processor.
- Register file organization.
- Pipeline depth.
- **Cache organization:** code size, type, replacement policy etc. as well as support for cache coherence.
- Communication interface.
- Number and types of functional units.

1.4.2 Interconnection Network

The simplest interconnection strategy is to use a single bus which is being shared by every other component for communication. Though this strategy is easy to implement, as the

number of processors go up, the bus becomes the bottleneck. It can be noticed that the shared bus must support multi master operation. All the components connected to this bus should tune their interfaces to use the bus protocol. Apart from this, designers have to implement some arbitration mechanism to resolve the conflicts.

There are several other interconnection choices which enhance the communication bandwidth at higher cost. These are: *multiple buses, multistage interconnection network (MINs) and crossbar switches*. Out of these, crossbar establishes one to one correspondence, but this is the most expensive.

The analyzed interconnection networks are quite general and their performance is application dependent. However, in an application specific multiprocessor architecture, one needs to analyze the possible communication traffic of the application. This analysis will lead to the decision whether one of the above interconnections is employed or a custom interconnection based on traffic is explored.

1.4.3 Memory Architecture

The processor speed has gone up many folds over the years, but memory speed though increasing has not kept pace. This is also due to the fact that higher complexity of the applications also demand larger memory sizes. Because of this speed mismatch, memory has become the bottleneck. One can notice that development of techniques such as caching, paging etc. are results of processor-memory speed mismatch. The problem becomes even more complicated in embedded systems because of real time processing requirements.

There are a number of factors which need to be considered in the design of memory architecture:

- Word size, line size and number of ports.
- Interleaved/non-interleaved memory and inter-leaving factor.
- Synchronous/Asynchronous memory.

- Pipelined/non-pipelined memory.
- Various transfer modes: sequential access, burst mode etc.

Various other memory configurations such as FIFOs, frame buffers, stream buffers, shared caches etc. can enhance performance many folds depending on application. Hence these memory configurations should also be explored. In essence there is a need for analyzing and choosing from a range of memory architectures to reduce the processor-memory speed mismatch to the minimum.

1.5 Application Specific MpSoC Design Subproblems

Figure 1.2 shows a typical MpSoC architecture. Designing such an architecture is a complex optimization problem. A designer has to decide on the following aspects manually or using CAD tools in order to realize an application specific MpSoC.

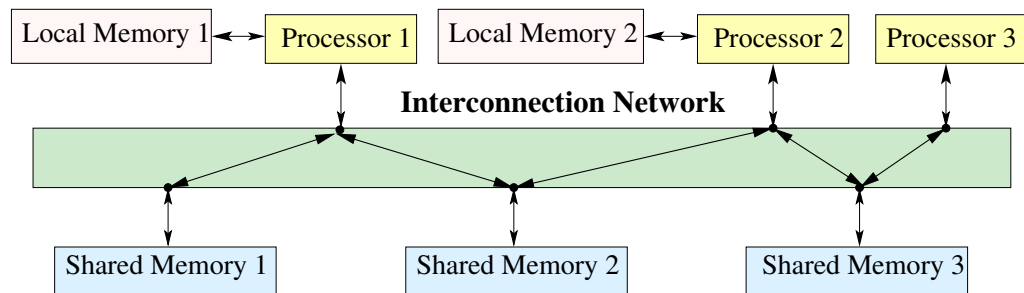


Figure 1.2: Typical MpSoC architecture

Architectural Resource Allocation: The architecture is composed of multiple processors connected to the memories. These processors could be of the same type (homogeneous multiprocessor) or of different types (heterogeneous multiprocessor). Architectural resource allocation is nothing but deciding the number and types of processors to be instantiated in the architecture. This phase is driven by application's

computation requirements. Apart from processor allocation, decision on memories is also taken.

Interconnection Network (IN) Design: Once processors and memories have been decided, they need to be connected through an interconnection network. The IN could be as simple as a shared bus, a cross-bar switch or some complex IN such as a mesh. This design is driven by the communication requirements of the application.

Application Mapping onto the Architecture: Mapping various parts of the application onto the architecture is another important design decision. Essentially, in this step, computation needs of the application are mapped to the processors and communication requirements to the communication resources such as buses, memories etc. This phase is again guided by the application's requirements and performance constraints.

Scheduling: There are two possible ways in which the application can be executed. One possibility is to statically schedule the application onto the architecture such that constraints are met. Static scheduling is mostly pessimistic and uses worst case behavior of the application in order to provide performance guarantees. The other possibility is to use a dynamic scheduler which takes scheduling decisions at run time. This scheme does not rely on worst case behavior and decisions are taken dynamically.

In rest of this Chapter, we discuss how the above problems have been handled in the past followed by an overview of our approach.

1.6 Previous Work

A lot of research has been done on various aspects of a general purpose parallel computer architecture. Both hardware and software problems have been studied extensively in this

domain [17] and several analytical and simulation based methods have been developed to assist architecture evaluation. In the following subsections, we will review these techniques and look at their applicability and shortcomings in application specific multiprocessor design.

1.6.1 Simulation Based Approaches

Because of the complexity of modeling a multiprocessor system, a large number of design methodologies are simulation based. There are various multiprocessor simulators for general purpose computing systems. MINT [76] is a toolkit on top of which multiprocessor memory hierarchy simulators can be built. This simulator works on MIPS platforms. Augmint [58] is an execution driven simulation toolkit for developing multiprocessor simulators for x86 platforms. RSIM [59] is also an execution driven multiprocessor simulator intended for ILP processors. ABSS [72] runs on SPARC hosts and retains an interface for MINT. Apart from these, there are several other multiprocessor simulators as well. Though these simulators cover a large design space, they are not suitable for application specific multiprocessors because of heterogeneous nature of these systems.

Another approach which has been investigated is core based design using co-simulation for verification [11, 48, 7, 81, 52, 8]. These models perform very detailed performance evaluations and are based on cycle accurate simulations. However, the main problem lies in generation of interfaces for different cores as different cores use different protocols for communication. Another drawback is that not much design space can be explored due to both the effort required in generating the model and time involved in performing the simulations.

The methods described above, take VHDL models of the COREs as input. The simulation time is large in this case. Because of large simulation times associated with HDL models, system level simulation methodologies have also been explored [23, 63]. Both of these use TSS (Tool for System Simulation) which is a cycle accurate simulator. Though

simulation time is relatively smaller than VHDL simulation, it is still large. Moreover, developing component models in this system is also very time consuming. Currently SystemC [73] is becoming popular and SystemC based models are also being explored for this purpose.

The approaches based on cycle accurate simulation as well as transaction level modeling (using SystemC etc.) have the potential to be used for performance prediction of a specific chosen design and/or selecting from a small set of architectural choices. These do not seem suitable for exploring a large design space.

There has also been some efforts to develop an alternative technique including Joshi et al. [37] for design space exploration which tries to exploit strengths of analytical approaches and simulation. This method relies on stochastic modeling of the application and performs simulation of this model which converges very fast. The method has been validated against commercially available simulators such as ARMulator and Cradle UMS simulator.

1.6.2 Analytical and Scheduling Based Approaches

Processor and multiprocessor modeling has been investigated using petrinets [64, 50, 80]. Petrinets were chosen because of their power to model concurrency effectively which is inherent in hardware systems. However, state space becomes very large for any real application which makes the model very difficult to analyze. The run-time behavior cannot be captured as well.

Amit Nandi et. al. [57] use *Stochastic Automata Network (SAN)* as a formalism for system level analysis. SANs are Markovian formalism belonging to the class of process algebras which are good at modeling communicating concurrent processes. Both the application and architecture are modeled as SAN. Mapping of application model onto the architecture model is done prior to analysis. This methodology is purely analytical and does not use simulation at any stage. Drawbacks of this methodology are simplified task graph and architectural models. Not many architectural choices can be explored. Further,

it is not suitable for ASIP based multiprocessor environment because of assumption of presence of pre-designed processors.

Bhuyan et. al. [12] compare analytical models of three main interconnection networks: crossbar switches, multistage interconnection networks and multiple bus systems. Though analytical models are available, they make certain assumptions about the system such as identical processors, uniform reference model and request independence. Uniform reference model implies that the destination address of a memory request is uniformly distributed among M memory modules which is generally not true. Similarly request independence assumption implies that request generated in a cycle is independent of the request in the previous cycle. In reality, this is an over simplification because a request that was rejected in the previous cycle will be resubmitted again.

A number of analytical models exist to evaluate effect of memory architecture on performance [28] as well. Most of these models are for interleaved memory and use queuing theory. They assume some request and service distribution and give expressions for memory bandwidth. These models also cannot capture dynamic behavior of the application. Apart from this, they also don't take into account factors such as access modes, number of ports, unequal memory cycle times etc.

Synthesis of optimal application specific multiprocessor architectures for DAG based periodic task graphs with real time processing requirements has been extensively studied from cost as well as power consumption points of view [19, 14, 47, 66]. The approaches for DAG based task graphs perform static scheduling and solve the synthesis problem which essentially consists of architectural resource allocation, binding of application components to architecture and scheduling. These approaches mainly suffer from the following limitations.

- Loops are represented as single node of the task graph derived from the application [40, 14, 75]. This makes the task graph unbalanced and limits the possible parallelism which could have been exploited. On the other hand, if we unroll the loops to fully

expose the application parallelism, it might lead to very large graphs for most real life applications, making the approach impractical.

- Effect of local communication is not properly modeled.
- Static scheduling is performed using worst case behavior. Typically, this leads to an over-design of the system.

1.6.3 Hardware Software Co-design Infrastructures

Ptolemy [42] is a design environment for concurrent, heterogeneous embedded systems. In Ptolemy, one can model the systems by using a model of computation such as Process Networks, Synchronous Dataflow Networks etc. These models are called domains in Ptolemy. The framework systematically allows to refine and simulate the system by making use of various code-generators. The environment has been designed in such a manner so as to extend it by adding new domains, and plugging in new analysis, synthesis and verification techniques.

Polis [8] was a hardware software co-design framework developed at University of California at Berkeley. The underlying model is Co-design Finite State Machine (CFSM) which can also be viewed as "Globally Asynchronous, Locally Synchronous" model of computation. Finite State Machines (FSM) can be extracted and directly given to FSM based verification algorithms. In Polis, hardware software partitioning is mostly interactive in which designers keeps on providing inputs. In this framework, after hardware and software synthesis, co-simulation is performed using Ptolemy.

Metropolis [9] provides an infrastructure based on a metamodel which is general enough to accommodate a variety of models of computation. Apart from capturing functionality, this metamodel can also capture architecture description and mapping information. The basic idea behind developing Metropolis is to provide an infrastructure in which new models of computation and new analysis and synthesis algorithms can be easily plugged-in.

Currently, in Metropolis, analysis can be performed using simulation and formal verification. Synthesis can be performed using Quasi-static scheduling (QSS) algorithm [16]. QSS accepts process network as input and creates static schedule in such a manner to reduce the context switch overhead. The underlying model is based on Petri-nets. In QSS, scheduling is not being performed under performance constraints. Further, as the underlying model is Petri-nets, it could result in a very large Petri-net for a reasonable size application which could seriously limit search choices considered.

In Ptolemy and Metropolis, what lacks is strong automatic synthesis of the architecture from the given application model and constraints. Our work is complementary to these infrastructures in the sense that we are providing an automatic synthesis framework for applications modeled as process networks.

Apart from co-design infrastructures discussed above, there are some other co-design environments as well which target specific sub-problems. One of the early work is Chinoch framework [15] in which focus is on interface synthesis. CoWare environment [65] developed at IMEC proposed single specification method required for different abstraction levels. In this work, authors have proposed systematic refinements, but no automatic synthesis is being done. In LYCOS co-synthesis environment [51], single processor and single ASIC has been chosen as the target architecture based on internal control dataflow graph representation.

1.6.4 MpSoC Synthesis for Process Networks

Another alternative is to start from the specifications, where application is modeled as process networks [38, 21, 70]. In these models, processes communicate through FIFO queues. Unlike task graphs, in process networks, computation and communication are interleaved and parallelism present in the application is made explicit.

If one is inclined to use DAG based task graph approach as discussed earlier, we note that the processes must be decomposed into sub-processes, but they need to be

mapped onto the same resource [14]. This requirement arises from the observation that sub-processes derived from the same process are tightly coupled in terms of sharing of variables etc. It suggests that the process network need not be further decomposed and its higher level characteristics can be used to synthesize the architecture for the given process network(s). We take this approach in our work.

In the past, researchers have addressed the problem of automatic synthesis of optimal architecture for process networks only partially. Dynamic scheduling of process networks has been addressed by Thomas Parks [60] and Basten et al. [10] on the given architectural resources. Leijten et al. [45] have proposed PROPHID multiprocessor architecture for process networks. This architecture essentially consists of a general purpose processor for control oriented processes, a number of application domain specific (ADS) processors and FIFO buffers for communications. Allocation of interconnection network (IN) bandwidth to various data streams has been addressed in [44]. In this work, architectural resource allocation and binding of application onto architecture is assumed to be pre-specified. PROPHID architecture does not exploit communication properties of application effectively to reduce interconnection network cost.

Piementel et al. [63] propose a co-simulation based approach for design space exploration of architectures for Kahn Process Networks (KPNs) [38]. In this methodology, architecture and application mapping is specified manually and co-simulation is used for performance analysis. A three stage process network refinement based approach has been proposed in [20] for heterogeneous multiprocessor mapping of process networks. In this approach, process network refinements are done manually to reduce communication overhead and the architecture is a fixed bus based architecture. Here, underlying architecture is a fixed bus based architecture which is specified manually and a dynamic scheduler is used to take care of scheduling of processes.

1.7 Our Approach

In this thesis, we address the automatic synthesis of optimal architectures for process networks in an integrated manner by developing a framework for this problem. In this framework, architectural resource allocation is performed in which resources (processors, memories etc.) are chosen from a component library. Alongwith resource allocation, we perform binding of various processes of the process network onto processors and queues onto memories. Here we assume that a dynamic scheduler will take care of run-time scheduling requirements. Another novel feature of our work is synthesis of interconnection network (IN) by exploiting communication patterns present within the application process network. As part of IN synthesis, we have also developed new analytical models to estimate the queuing delays in heterogeneous multiprocessors with non-uniform memory accesses. While performing validation, we found that there are no benchmarks available with applications modeled as process network. Further, developing such benchmarks for real applications is very time consuming process and not easily portable across different PN frameworks. Hence, we developed a new random process network generator (RPNG) which can create sample process networks with defined characteristics quickly and aid in quick validation and comparison of MpSoC synthesis techniques.

1.8 Thesis Outline

The outline of rest of the thesis is as follows. The overall methodology is described in Chapter 2. We discuss analytical models to estimate queuing delays in heterogeneous multiprocessors in Chapter 3, followed by MpSoC synthesis problem formulation in Chapter 4. In Chapter 5, we describe MpSoC synthesis heuristic in detail. Random Process Network Generator (RPNG) is discussed in Chapter 6. Experimental results alongwith discussion is given in Chapter 7. Finally, in Chapter 8, we summarize our contributions and discuss future works.

Chapter 2

Overall Methodology

In this Chapter, we discuss the overall synthesis methodology. Figure 2.1 shows the *Srijan Synthesis*, a hardware-software codesign framework. This thesis forms part of *Srijan* framework. Hardware and software estimates are generated from the application specification and component library. These estimates are used to explore the multiprocessor architecture design space and identify a specific architecture alongwith the application mapping onto it. After deciding the architecture and mapping of the given application onto it, the code is generated for the parts of the application which are mapped onto processor(s) and RTL is produced for the parts of application mapped onto hardware. In this work, we focus on the system-level design space exploration and solve the problem of synthesis of application specific multiprocessor architectures when the initial application specification is in the form of process networks.

2.1 System Model

2.1.1 Models of Computation

The basic objective of this thesis was to develop a synthesis framework for Multiprocessor System-on-Chip (MpSoC) architectures for large media processing applications. An

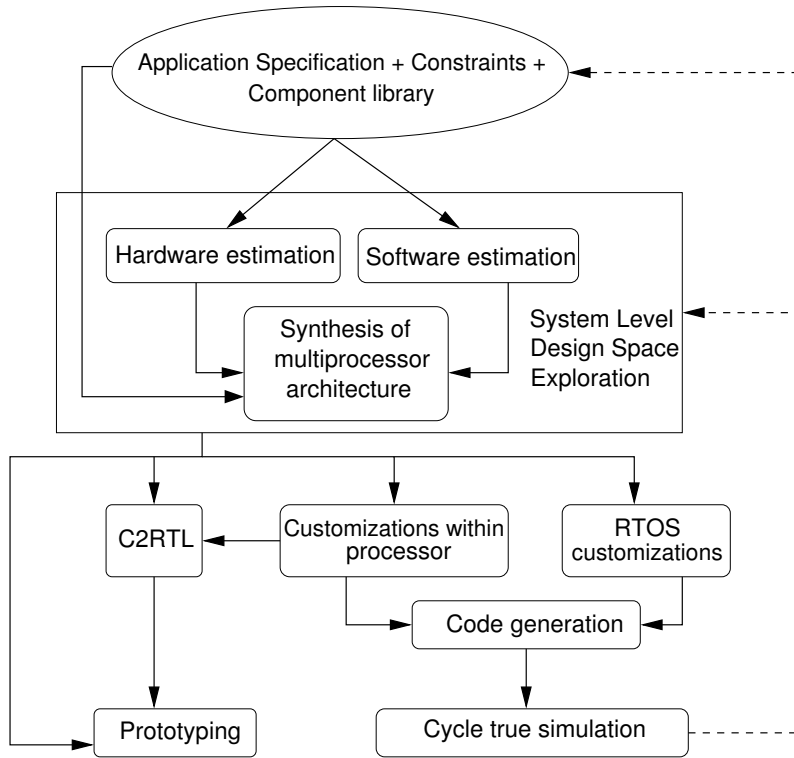


Figure 2.1: Srijan synthesis framework

application model should have the following characteristics so as to best suit the purpose.

- Allows to model parallelism and streaming.
- The application model should be compositional which allows to compose bigger applications by composing models of smaller applications.
- It should lead to a compact representation to make the analysis and synthesis manageable for large problems.
- The application model should be suitable for platform based design so that fast turnaround time for the design can be achieved.

A large number of modeling techniques have been proposed in the literature. All these models have their own advantages and disadvantages making it quite difficult to choose

the suitable model for the given applications. We discuss some of widely studied modeling techniques next.

Petri Nets

One of the widely studied modeling technique is Petri-nets which was first proposed by Petri in his Ph.D. thesis [62]. Petri Nets are used for describing and studying systems that are characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic and/or stochastic. There are two kinds of nodes in Petri-nets: places and transitions. Arcs are either from a place to a transition or a transition to a place. In modeling, places represent conditions and transitions represent events. A marking (state) assigns to each place a non-negative integer. If a marking assigns a place p with a non-negative integer k we say that “ p is marked with k tokens”. The presence of a token in a place is interpreted as holding the truth condition related to that place. After basic Petri-nets as discussed above, a number of variants of Petri-nets have been proposed in the literature [61, 56, 82] to extend the scope of modeling and address associated problems.

The basic problem with Petri Nets is that even a small system needs many states and transitions. This problem is known as the “state explosion”. Because of “state explosion” problem, in practice, Petri-nets have not been found suitable for architectural synthesis.

Process Networks

Another model which is widely studied and applied to system design is Process Networks. This is a Model of Computation which was originally developed for modeling distributed systems, but has been found very convenient for modeling signal processing systems. Process Networks are natural model for describing signal processing systems where infinite streams of data are incrementally transformed by processes executing in sequence or parallel. Process Networks were first introduced by G. Kahn [38] (Kahn Process Networks - KPN) and then widely studied later. One such example being [60].

KPNs are directed graphs where nodes represent Processes and arcs are infinite FIFO queues that model channels connecting these processes. Writing to a channel is non blocking, but reading is blocking. If a process tries to read from an empty input, it is suspended until it has enough input data and the execution context is switched to another process. A process may not “test” channel for presence of data. At any given point a process, is either “active” or “blocked” waiting for data on one of its channels. Each process is a sequential program that consumes tokens from its input queues and produces tokens to the output queues. Each queue has one source and one destination.

There are the following set of properties which make Process Networks suitable as a Model of Computation for data dominated media processing applications.

- The network has no global data.
- Concurrency can be implemented safely.
- Scheduling is transparent to the user.
- Compositional: Hierarchy and scalability.
- It is deterministic.

Drawback of process networks is that they cannot model non-deterministic behavior and that is why they are not suitable for modeling control dominated reactive embedded systems.

Dataflow Process Networks

Another model studied in the literature is Dataflow Networks by Edward A. Lee et al. [43]. In this model, arcs represent the FIFO queues, but now the nodes of the graph, instead of representing processes, represent actors. Instead of responding to the simple blocking-read semantics of Process Networks, actors use firing rules that specify how many tokens must

be available on every input for the actor to fire. When an actor fires, it consumes a finite number of tokens and produces a finite number of output tokens. A process can be formed by repeated firings of a dataflow actor. Synchronous Dataflow Networks (SDF) is a special case of Dataflow Networks in which the number of tokens consumed and produced by an actor is known before the execution begins. The same behavior repeats in a particular actor every time it is fired. Arcs can have initial tokens. The advantage with SDFs is that they can be statically scheduled. The drawback of Dataflow Networks is that they are specified at much finer level than Process Networks which, in practice, could become very large for large media processing applications.

DAG Based Periodic Task Graphs

Synthesis of optimal application specific multiprocessor architectures for DAG based periodic task graphs with real time processing requirements has been extensively studied from cost as well as power consumption points of view. In this model, nodes represent computation and arcs represent communication. This model mainly suffers from the following limitations.

- Loops are represented as single node of the task graph derived from the application. This makes the task graph unbalanced and limits the possible parallelism which could have been exploited. On the other hand, if we unroll the loops to fully expose the application parallelism, it might lead to very large graphs for most real life applications, making the approach impractical.
- Effect of local communication is not properly modeled.

Based on the above discussion, we realized that Process Networks are best suited for our MpSoC synthesis problem.

2.1.2 Application Model

As shown in Figure 2.2, the application is specified in the form of a process network. Processes communicate with each other using FIFO queues and computation and communication within any process is interleaved. All the processes are assumed to be periodic. In each iteration of their invocation, they produce or consume certain number of tokens on queues connected to them. We further assume that the size of a token being communicated on any queue always remains the same and the size of each queue (maximum number of tokens within it) is known apriori. We note that once the size of each queue is specified and the queue is full, then the writes on that queue become blocking. We further assume that a dynamic scheduler will take care of the run time scheduling requirements of the mapped process network.

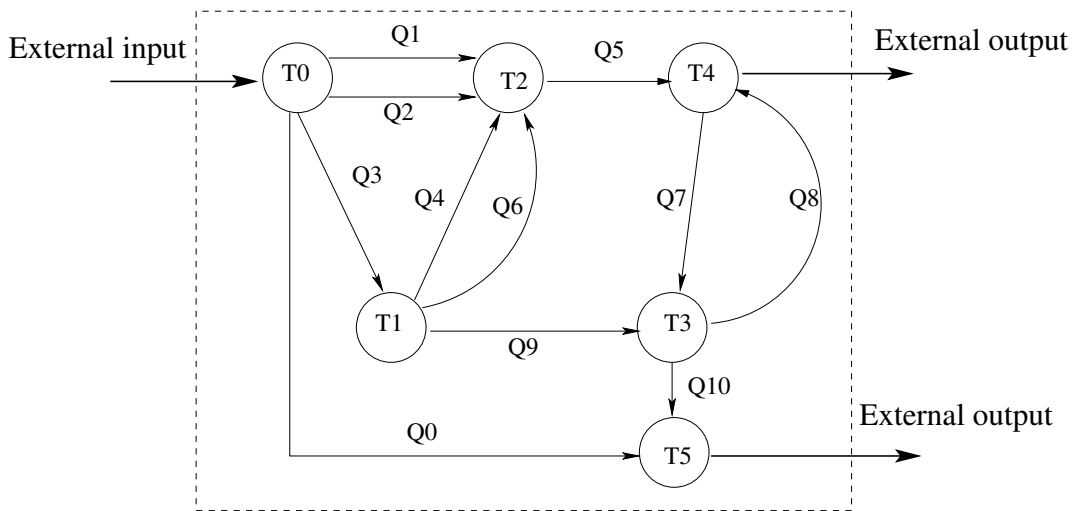


Figure 2.2: Example process network

The real time constraint is specified in terms of how often data tokens need to be produced/consumed on/from queues. We represent this as throughput constraints on queues which is nothing but the number of tokens to be produced/consumed per second. In turn, these throughput requirements on the queues impose processing requirements on corresponding reader and writer processes.

2.1.3 Architectural Component Library

Our synthesis algorithm (described in Chapter 5), tries to minimize cost and/or energy and improve resource utilization. This results in architectures of the type shown in Figure 2.3, consisting of a set of processors along with their local memories, shared memories and interconnection network. Our architectural component library contains modules corresponding to each of these. A processor could be a non-programmable unit like an ASIC or a programmable unit such as a RISC or DSP processor. A processor has a set of attributes. These are: cost of the processor, different voltage levels supported by the processor (and corresponding clock periods) and the number of cycles taken by the processor during a context switch. Apart from these, we also specify the number of cycles taken and energy consumed at different voltages by a process when mapped onto the processor for one iteration. These values represent exclusive process-processor mapping and thus do not consider memory conflicts and context switch overheads.

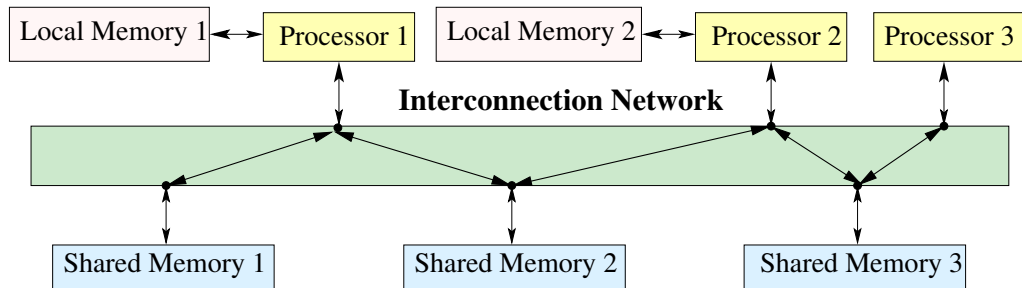


Figure 2.3: Sample MpSoC architecture

There are local memory modules along with each processor and shared memory modules accessed by multiple processors. We characterize these memories in terms of their base cost, bandwidth and the energy consumed in the transfer of a single word. The base cost is the equivalent hardware cost of allocating a single word in corresponding memory. Here we assume that there is no limit on the size of any memory.

The interconnection network of the synthesized architecture is a partial cross-bar as

shown in Figure 2.3. Typically these switches are implemented using multiplexers having cost which depends on its size (number of sources). Hence, to take care of interconnection cost, the component library also defines cost of each link. Further, the library also defines the energy consumed in transferring a single word on these links.

2.2 Hardware Estimation

2.2.1 Hardware Estimator Tool

Hardware estimation is required to get an estimate of clock period, area and bounds on execution time when some part of the application might be mapped onto dedicated hardware. Getting HW metrics using estimation is much faster than actual synthesis. Such estimation approach is necessary for providing values of HW metrics during the iterative design space exploration (DSE) process. We have used the hardware estimation flow described in [39] for this purpose. This flow works in two stages. In the first stage, the clock period is estimated. In the second stage, upperbound on execution time is estimated using a fast estimation algorithm.

The *Clock Slack Minimization (CSM)* method [13] estimates the “optimum” clock by minimizing the average slack in the entire data flow graph of the program. This method does not consider the critical path which results in longer execution times. The clock estimation described in [39] takes into account the critical paths to determine optimal clock period. This algorithm is termed as *Critical Path Slack Minimization (CPSM)* algorithm.

After computing clock period using CPSM, we estimate the upper bound on execution time using *Resource Use Method (RUM)* described in [39]. RUM considers dependencies of the operations in the application and hardware resources to get an estimate of the execution time. This algorithm also takes into account the constraints due to limited number of ports in the register file and memory. The complexity of this algorithm is linear which makes

it suitable to quickly evaluate various hardware implementations for different resource constraints.

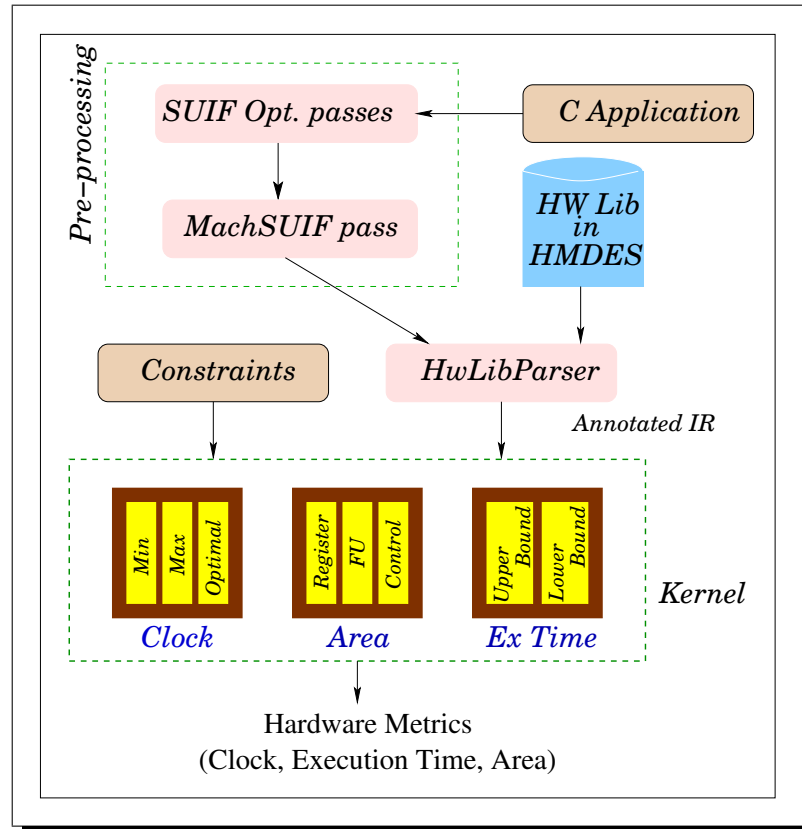


Figure 2.4: *HwEst*: Hardware Estimator Tool

We implemented the hardware estimation flow discussed above as shown in Figure 2.4. In this implementation, an application written in C language is pre-processed to generate the program’s SUIF2 [78] intermediate representation (IR). Then classical optimizations such as common sub-expression elimination (CSE) available in SUIF2 are applied on the IR. The MachSUIF [49] passes are used to generate the control flow graph (CFG) of the program and data dependence graph (DDG) at the basic block level. The *HwLibParser* pass then annotates the library information in the IR. We used HMDES [33] to describe our parameterizable resource library. The annotated IR is fed to the next stage along with the constraints set. This stage consists of independent passes, each estimating a particular metric Clock, Area and Lower and upper bounds on execution time.

2.2.2 Validation of Hardware Estimator Tool

We developed synthesizable RTL descriptions (in VHDL) of the benchmarks taken from Mediabench [41], Mibench [54] suite. We used Synopsys Design Compiler (DC) [4] to synthesize the HDL descriptions using LSI 10K technology libraries to obtain metrics like critical path length, and chip area. We validated our estimates with the synthesized values. Table 2.1 presents a quantitative comparison between the *HwEst*'s upperbound estimates and synthesized values of the kernel of each benchmark. In these experiments, we assume 4 read and 4 write ports in register file and memory. The first column gives the names of different benchmarks used. The next three columns, give the execution times in terms of number of cycles and then in "ns" generated by *hwEst* and Synopsys DC. We note that our bounds are 17-35% off from the synthesized values. However, such deviation is expected because it is an upperbound and also generated at a much higher level of abstraction. In spite of that, its utility in pruning the design space is invaluable. Moreover, the lower run time (by over two orders of magnitude) per exploration cycle makes *HwEst*-based performance estimation of candidate architectures ideal for rapid DSE.

2.3 Software Estimation

Before deciding the architecture, we need to have an estimate of the number of clock cycles taken and energy consumed by various parts of the application on a specific processor present in the component library. We used the methodology proposed by Manoj Jain et al. [36] to estimate the number of clock cycles taken by the application onto the processor under consideration. The reason to choose this methodology was that in their results, they have shown the estimated values to be as close as 10%. Moreover, they also showed that their tool runs almost 80 times faster than a simulator based approach.

Further, to estimate the energy consumed by some part of the application, we make use of results from JouleTrack [67]. Here, the authors have shown that the energy consumed by

benchmark	<i>hwest</i>		DC (ns)	% error	runtime (s)	
	(cycles)	(ns)			<i>hwest</i>	DC
adpcm coder	47	235	200.64	17.13	5.17	636.1
adpcm decoder	29	145	120.69	16.77	4.25	390.4
elliptic wave filter	25	125	94.85	24.12	3.71	380.1
pipelined fir filter	27	135	95.67	29.13	3.73	370.6
wavelet	37	185	145.41	21.4	4.41	577.9
kalman filter	18	90	73.5	22.69	3.61	373.1
differential equation	31	155	115.12	25.73	4.25	420.3
fft	33	165	140.86	22.69	4.3	456.3
cubic	17	85	63.75	34.8	3.29	390.4

Table 2.1: validation of upperbound estimates

the software running on a processor depends primarily on the instructions executed and are not particularly sensitive to the instruction sequence. Apart from energy consumed within processors, we also need to get an estimate of energy consumed in the interconnection network. We use results presented by Wang et al. [77] for this purpose. Hence, as part of software estimation, we extended the methodology proposed by Manoj et al. [36] by making use of the above results so that performance as well as energy numbers for various parts of the application can be obtained.

Figure 2.5 shows the overall software estimation flow. We first convert the application written in 'C' language to SUIF2 [78] IR. Then, this form is converted to MachSUIF [49] IR. Dependence analysis is performed to extract the function level control flow graph and basic block level data flow graph. Then, for each basic block, local register allocation using register reuse chains is performed followed by the priority based list scheduling. Latency of this schedule serves as the execution estimate of the basic block. This estimate multiplied with the profile information gives the local estimates. Now, local execution estimates are

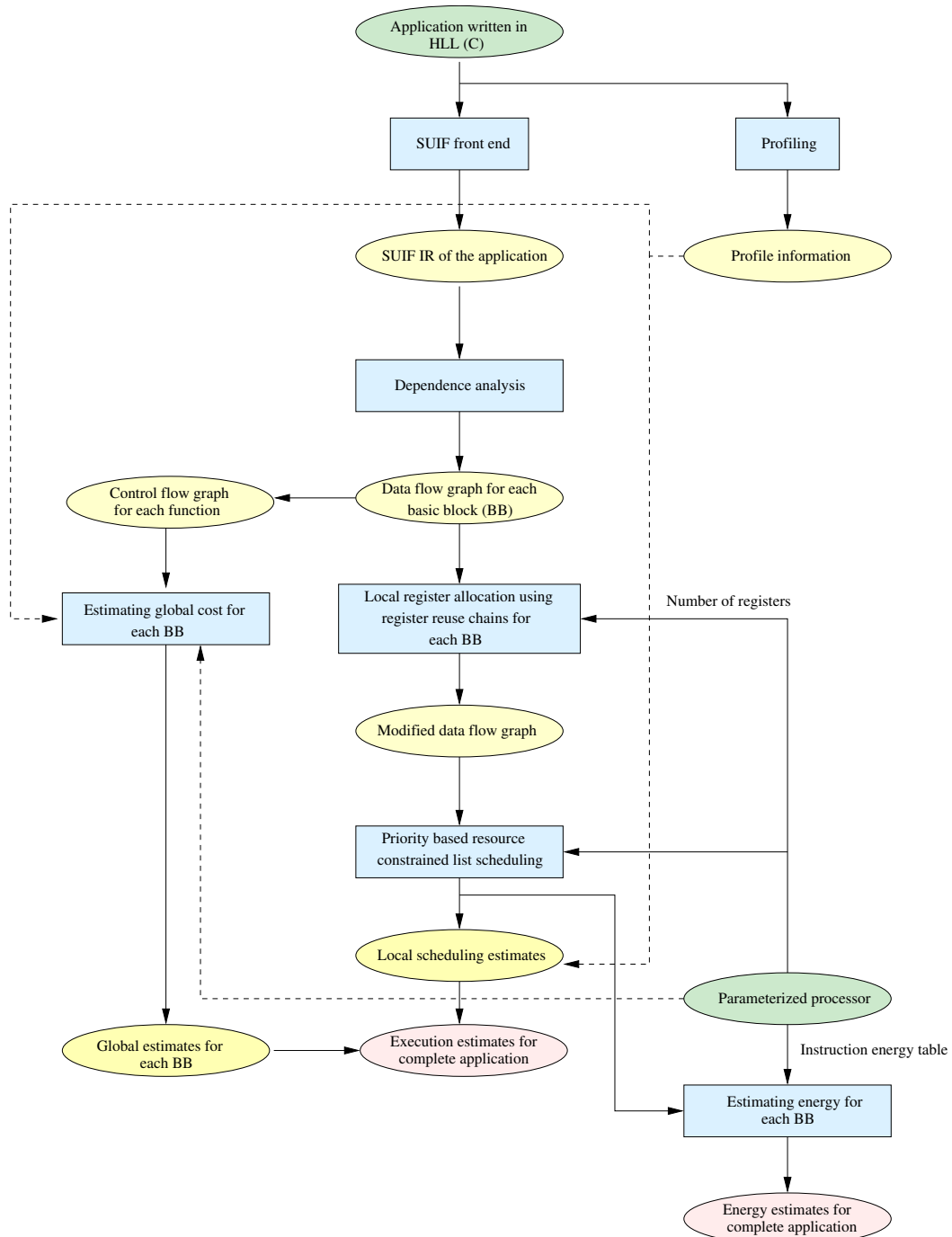


Figure 2.5: Software Estimation Methodology

[36]

added to the global cost to get the final estimate for the total execution time. Energy estimates of the basic blocks are obtained by making use of the basic block level schedules

and the instruction energy table.

2.4 Multiprocessor Architecture Synthesis

Once software and hardware estimates are available for the given application and a component library, synthesis of the architecture consists of the following tasks to minimize total cost or energy.

1. Allocation of processors, local and shared memories
2. Binding of processes to processors and queues to local or shared memories
3. Allocation of communication components such as buses

Figure 2.6 shows an instance of a synthesized architecture. In this example, the application process network is composed of 3 processes and 3 queues. The synthesized architecture consists of 2 processors, 1 local memory and 1 shared memory. *Queue 1* is mapped to the local memory of *Processor 1* because reader and writer processes of *Queue 1* are mapped here. On the other hand *Queue 2* and *Queue 3* are mapped to the shared memory as their reader and writer processes are mapped to two different processors.

In Figure 2.6, we observe that if more than one queues are mapped to the same shared memory module, queuing delays will occur due to simultaneous request for data on these queues. Hence, we must make proper estimation of these queuing delays, when mapping of queues onto memories is being performed. In Chapter 3, we present estimation models to estimate queuing delays for burst as well as non-burst mode communication traffic. Later in Chapter 4, we also describe, how these models are used during MpSoC synthesis.

We note that the following considerations help us to reduce the cost and energy of the synthesized architecture.

- Mapping a process onto a low cost, low energy processor.

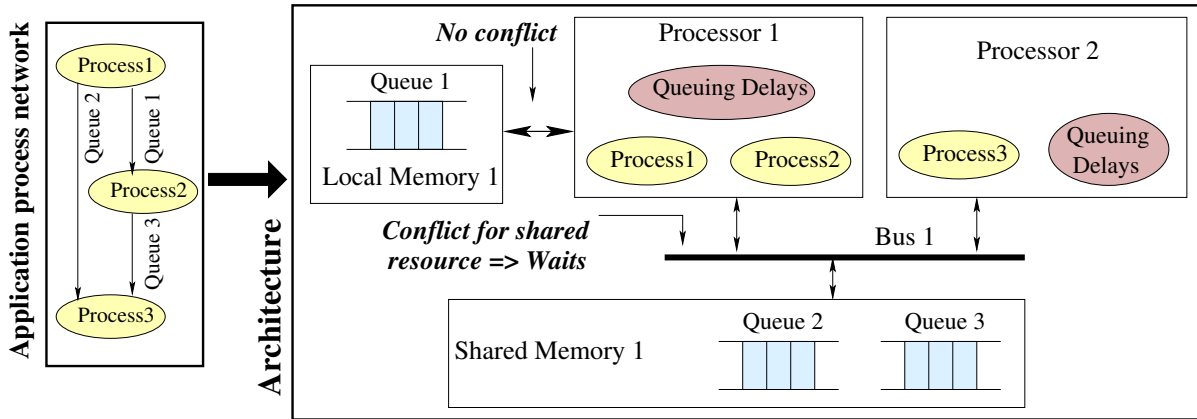


Figure 2.6: Example illustrating MpSoC synthesis for application modeled as process network

- Mapping densely connected processes to the same processor will result in putting more queues in the local memory. This in turn helps to reduce shared memory as well as interconnection network (IN).
- Mapping frequently accessed queues onto the local memories will help to reduce conflicts at shared memories. It will allow more queues to share the same shared memory resulting in lower IN cost.

Based on above considerations, we propose an MILP and a heuristic based solution in Chapter 5 for the synthesis problem outlined above. In heuristic based approach, we construct the solution by employing a fast dynamic programming based technique. Our approach not only allows it to be used in a design space exploration loop, but also provides a good initial solution for an iterative refinement phase.

2.5 Summary

In this Chapter, we have outlined the overall methodology employed for MpSoC synthesis. The methodology accepts application represented as process networks and an architectural

component library as inputs. Software and hardware estimates are generated for various potential mappings of the application onto the architecture. These estimates are used to drive the design space exploration for MpSoC synthesis.

Chapter 3

Estimation of Queuing Delays for Heterogeneous Multiprocessors

As we discussed in the previous Chapters, many applications such as streaming applications (MPEG2, JPEG etc.) offer large amount of parallelism. They also impose real time processing demands. To exploit parallelism present in these applications and satisfy their processing requirements, multiprocessor architectures are employed. To meet constraints such as cost, area, power consumption etc., the architecture is heavily customized which makes it heterogeneous having non-uniform memory access.

In this Chapter, we present analytical models to estimate queuing delays for heterogeneous multiprocessors with non-uniform memory access for burst as well as non-burst traffic. The outline of the chapter is as follows. Related work is discussed in Section 3.1 followed by description of system architecture in Section 3.2. Estimation models for burst mode traffic is described in Section 3.3 and for non-burst mode traffic in Section 3.4. Some experimental results validating estimation models are discussed in Section 3.5 followed by summary of the chapter in Section 3.6.

3.1 Related Work

Analytical estimation models for homogeneous multiprocessors with uniform memory access has been studied extensively in the past [53, 35, 34, 29]. These studies have been done for many interconnection architectures such as single shared bus, multiple bus and cross-bar switches. In this, request rate of each processor has been assumed to be the same and processors access different memories with equal probability. Various solution methods have been used to solve the underlying queuing network to compute resource utilization.

Unlike homogeneous multiprocessors, heterogeneous multiprocessors have not been studied extensively. Marco and Mario [53] relax the equal access rate of processors to memory by assuming that one of the processors generates a fraction of total memory traffic and rest of the processors equally share the remaining fraction of memory traffic. In this work, it has also been assumed that one memory is accessed from any processor with a certain rate and the rest of the memories are accessed uniformly. This model is called *favorite memory* model. *Favorite memory* access model has also been dealt with in [18, 34], but there it has been assumed that the processors have equal access rates to the memories.

Don Towsley [74] gives a solution for the general case of heterogeneous multiple bus based multiprocessor architecture. Processors are assumed to have different access rates to the memories with different access probabilities. In this work, the surrogate delay approach has been employed to solve the queuing network. In this model, a surrogate delay which represents the mean bus queuing delay gets added to the mean memory service time.

Akiri Fukuda [29] has presented an estimation model for memory interference for cross-bar switch based heterogeneous multiprocessor architecture. Solution approach called *Equilibrium Point Analysis (EPA)* has been used to solve the underlying queuing network. The basic idea in this approach is that in the steady state, number of processors going from computing mode (P) to memory mode (M) is equal. Equilibrium point is derived and it is used to compute utilizations. Recently, performance models for mesh interconnected shared

memory heterogeneous multiprocessor has been proposed in [68] using Approximate Mean Value Analysis (AMVA) technique [26].

The estimation models presented in [74, 29, 68] have the following limitations.

- Memory access request rate of a processor in the architecture doesn't get modified because of blocking of an earlier request.
- As shown in [32], mean bus delay is not only the function of access rates of the processors but also the arbitration policy. This has not been taken into account in the presented estimation models.

3.2 System Architecture

The target architectures considered are shown in Figure 3.1. Architecture shown in Figure 3.1(b) reduces to that of Figure 3.1(a) for single memory case. For brevity, only 3 processors have been shown in these figures, but this number is extendable.

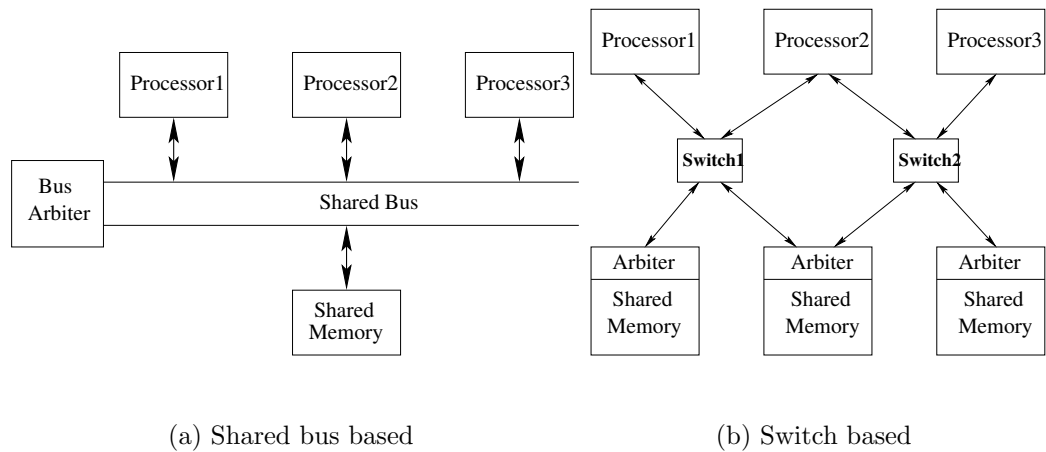


Figure 3.1: Architectures considered

One of the considerations is that of arbitration policy. There are various possibilities such as static priority scheme, Earliest Deadline First (EDF) scheme etc. In the static

priority scheme, fixed priority is pre-assigned to all the processors asking for access to memories. The problem with this scheme is that it might lead to starvation of some of the low priority processors.

In the EDF scheduling scheme, in case of contention, higher priority is given to the component which requires lesser service time. After the service is completed, priority of that particular component, is assigned to minimum. Since the priority of the component asking for the service keeps on getting changed, this scheme also falls in the rotating priority class. Further, as information of service time is required in EDF scheme, it is applicable only for the burst mode traffic. In general, it has been observed that "rotating priority" schemes perform well and provide fair scheduling. This is the reason why we have assumed this priority scheme in our analysis.

Accesses to the memories from the processors have the following properties:

1. Processors need not be identical and they access the memories with different rates. Hence the architecture is heterogeneous and memory accesses are not uniform.
2. A processor need not access all the memories.
3. An access request which is denied due resource contention is submitted again. Except resubmitted access requests, initial requests are assumed to be data independent for each processor.
4. The arbitration policy has been assumed to be "rotating priority".

Next we describe how queuing delays are computed for burst and non-burst communication in Sections 3.3 and 3.4 respectively for the architectures shown in Figure 3.1. Let us also recall that in burst mode, control of the bus is not released until all the data buffered by the requesting device has been transferred to/from memory. Whereas in non-burst communication, the bus is released after each transfer. This mode requires that the

bus request/acknowledge sequence is performed for every transfer. This overhead can result in a drop in overall system throughput if a lot of data needs to be transferred.

3.3 Analysis of Burst Mode Traffic

If more than one channel is mapped onto the same memory module, it is possible that while some request on one channel is in service, another request on the other channel arrives. Figure 3.2 shows one such interference between requests on two channels Q_1 and Q_2 . In Figure 3.2, D_1 and D_2 are the average number of time units between two successive data requests on channels Q_1 and Q_2 respectively. Similarly, the sizes of requests (token sizes) on Q_1 and Q_2 are sz_1 and sz_2 respectively. In this example, during one successive requests on channel Q_1 , we have used the discrete time scale $0 \leq t \leq D_1$ to denote the time instant at which a request on channel Q_2 arrives with respect to the request on channel Q_1 . Similar scale can be defined for Q_2 as well.

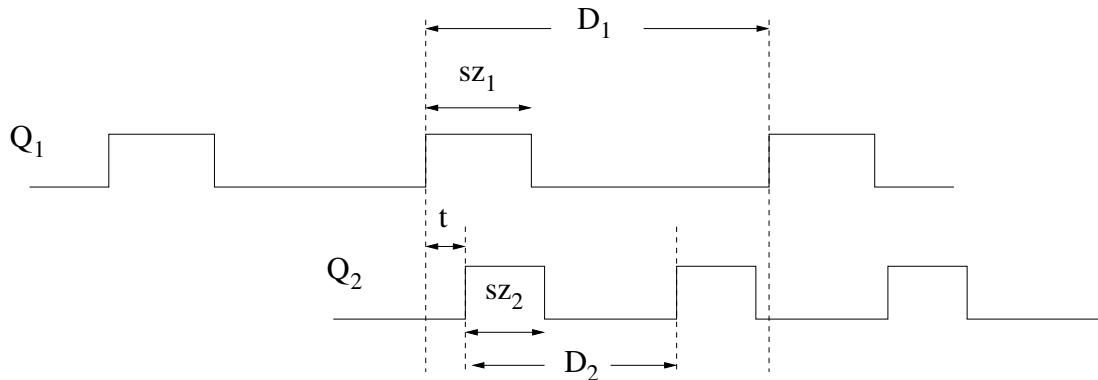


Figure 3.2: Mutual interference of read/write requests on queues Q_1 and Q_2

We make the following assumptions about the data requests.

- For a particular request on channel Q_2 , there is an equal probability of its occurring at any instant of time between two successive requests on Q_1 and vice-versa.
- Data transfer occurs in burst mode and transfer delays are proportional to the size of the data transfer request.

Hence, the probability p_{2t} of a request on Q_2 between two successive requests of Q_1 at an instant $t \leq D_1$ is $\frac{1}{D_1}$. Now, the delay caused on Q_2 when request on it arrives at instant t is as follows.

$$d_{2t} = \begin{cases} 0 & \text{if } t > sz_1 \\ sz_1 - t & \text{otherwise} \end{cases}$$

Now expected delay caused on channel Q_2 because of Q_1 is:

$$Ed_2 = \sum_{t=0}^{D_1} (p_{2t} \times d_{2t}) = \sum_{t=0}^{sz_1} \left(\frac{1}{D_1} \times (sz_1 - t) \right) = \frac{sz_1 \times (sz_1 + 1)}{2 \times D_1}$$

Some refinement is required as Q_1 will also see some delay because of mutual interaction.

$$Ed_2 = \frac{sz_1 \times (sz_1 + 1)}{2 \times (D_1 + Ed_1)} \quad \text{and} \quad Ed_1 = \frac{sz_2 \times (sz_2 + 1)}{2 \times (D_2 + Ed_2)}$$

Since, more than one channel might be mapped onto the same memory module, any channel on this memory will be in a possible conflict with others. Hence, total delay caused on channel Q_r because of mutual interactions of remaining channels will be the summation of delay caused by a single channel.

$$Ed_r = \sum_{s|r \neq s} \frac{sz_s \times (sz_s + 1)}{2 \times (D_s + Ed_s)} \quad (3.1)$$

Now equation 3.1 can be generalized to handle read-read, write-read, read-write and write-write conflict cases. $rrcon_{rspq}$ is the delay seen by Q_r due to interference of Q_s when access on Q_s is being made from processor PR_p and on Q_r from processor PR_q for reads. Similarly, $rwcon_{rspq}$, $wrcon_{rspq}$ and $wwcon_{rspq}$ correspond to read-write, write-read and write-write delays respectively.

We note that the reader and writer of a channel might get mapped onto different processors operating at different speeds. This would result in different access rates for the channel mapped onto a specific memory. This is the reason why we need to distinguish between read and write accesses. $rnpr_p$ is the number of time units between two consecutive read accesses on Q_r from PR_p . Similarly wnp_{rp} is the number of time units between two

consecutive write accesses. Let $qwr(r)$ and $qrd(r)$ be the indices of the writer and reader processes of channel Q_r respectively. If ncy_{ip} is the number of cycles taken by the process T_i when mapped onto processor PR_p for one iteration and $ntkn_{ir}$ is number of tokens produced/consumed per iteration by T_i on/from Q_r , then $rnpr_p$ and $wnpr_p$ can be obtained as follows.

$$rnpr_p = \frac{ncy_{qrd(r),p}}{ntkn_{qrd(r),r}} \quad \text{and} \quad wnpr_p = \frac{ncy_{qwr(r),p}}{ntkn_{qwr(r),r}}$$

From previous analysis, generalized equations become:

$$rrcon_{srqp} = \frac{sz_r \times (sz_r + 1)}{2 \times (rnpr_p + rrcon_{rspq})} \quad (3.2)$$

Similar equations can be written for other kinds of memory accesses discussed above. Now, these simultaneous equations are solved iteratively. During experimentations, we observed that the convergence occurs in the first few iterations. Delays computed from these equations are to be reflected in each access delay.

3.4 Analysis of Non-Burst Mode Traffic

Hwang and Briggs [35] proposed a contention model for shared bus with request resubmission as shown in Figure 3.3. The processor remains in the accepted state A by either making a request for the bus and getting it accepted or not making a request. It goes to the waiting state W, if its request is denied. Processor remains in W state and keeps on resubmitting the request unless its request gets accepted. Let P_a be the probability that the request will get accepted. In Figure 3.3, r is the probability that processor makes a request for the bus. If probability of being in state A is q_A and probability of being in state W is q_W , then state probabilities are found to be as follows.

$$q_A = \frac{P_a}{P_a + r \times (1 - P_a)} \quad \text{and} \quad q_W = 1 - q_A$$

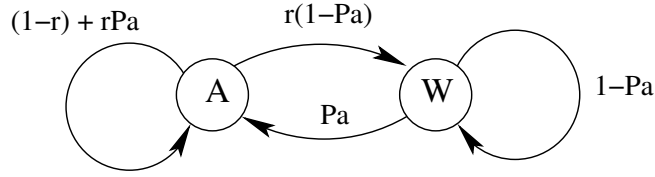


Figure 3.3: Two state request resubmission model

The actual offered request rate because of resubmission is:

$$r' = r \times q_A + q_W = \frac{r}{P_a + r \times (1 - P_a)} \quad (3.3)$$

The probability that a processor makes a request and it gets accepted is P_a . If the number of processors are n and they make request uniformly, then the probability that none of the processors are making request is $(1 - r')^n$. Hence, the request acceptance probability can be obtained as follows.

$$r \times P_a = \frac{1 - (1 - r')^n}{n} \Rightarrow P_a = \frac{1 - (1 - r')^n}{n \times r} \quad (3.4)$$

Equation 3.3 and 3.4 can be solved iteratively to compute the value of r' . The bandwidth offered by the bus can be derived from this.

In the model shown in Figure 3.3, there is one hidden state in which processor performs normal computation and does not access the shared resource. We term this state as *computing state C*. After exposing this state, we extend the request resubmission model proposed above for heterogeneous multiprocessor case.

3.4.1 Single memory case

We assume that there are a total of n processors. r_i is the request rate and P_{a_i} is the probability of request getting accepted for the i^{th} processor. Various states of a processor have been shown in Figure 3.4. State C_i is the computing state when the processor performs the normal computation and does not make any request. Processor goes to the accepted state A_i when it makes a request and it gets accepted, otherwise it goes to the waiting state

W_i . Processor keeps on resubmitting the request unless it gets accepted. In W_i processor is blocked. Processor comes back to the computing state C_i when there is no new request.

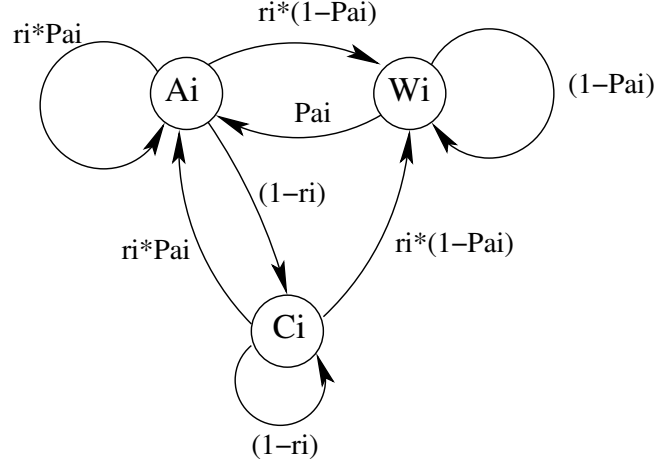


Figure 3.4: States of a processor

State transition matrix for i^{th} processor for state vector $(C_i, A_i, W_i)^T$ is as follows.

$$S_i = \begin{pmatrix} 1 - r_i & r_i \times P_{a_i} & r_i \times (1 - P_{a_i}) \\ 1 - r_i & r_i \times P_{a_i} & r_i \times (1 - P_{a_i}) \\ 0 & P_{a_i} & 1 - P_{a_i} \end{pmatrix}$$

Let state probabilities be q_{C_i} for state C_i , q_{A_i} for state A_i and q_{W_i} for state W_i . After solving the Markov model of the above transition matrix with boundary condition $q_{C_i} + q_{A_i} + q_{W_i} = 1$, state probabilities are found to be:

$$q_{C_i} = \frac{(1 - r_i) \times P_{a_i}}{P_{a_i} + r_i \times (1 - P_{a_i})} \quad \text{and} \quad q_{A_i} = \frac{r_i \times P_{a_i}}{P_{a_i} + r_i \times (1 - P_{a_i})}$$

$$\text{and} \quad q_{W_i} = 1 - (q_{C_i} + q_{A_i})$$

Looking at the state probabilities, one can notice that the state diagram of Figure 3.4 is equivalent to that of Figure 3.3. A processor makes a request with rate r_i from states C_i and A_i . Apart from this, processor always makes a new request from state W_i . Hence modified request rates for all the processors can be written as:

$$\forall i : \quad r'_i = r_i \times (q_{C_i} + q_{A_i}) + q_{W_i} = \frac{r_i}{P_{a_i} + r_i \times (1 - P_{a_i})} \quad (3.5)$$

Acceptance probability of a request from a processor is a function of arbitration policy. As *Round Robin (rotating priority)* arbitration policy has been found to perform quite well, we assume that the same is being used. The highest priority processor will have its priority equal to 1. In [32], probability of successfully accessing the bus has been derived for various arbitration protocols for multiprocessor architectures having uniform memory access. We perform similar analysis for non-uniform case for the rotating priority protocol.

We consider the following two variations of the rotating priority arbitration.

Case 1: As soon as request of a processor is accepted, its priority becomes the lowest.

Here cyclic processor sequence in the priority queue is not maintained. For example if the initial priority queue of the processors is $\{PR_1, PR_2, PR_3, PR_4\}$ and request of processor PR_2 gets accepted, then in this case, the new priority queue would be $\{PR_1, PR_3, PR_4, PR_2\}$.

Case 2: On acceptance of a processor's request, priority of all the processors is shifted such that this processor's priority becomes the lowest. Here initial cyclic ordering of processors in the priority queue is maintained. For example if the initial priority queue of the processors is $\{PR_1, PR_2, PR_3, PR_4\}$ and request of processor PR_2 gets accepted, then in this case, the new priority queue would be $\{PR_3, PR_4, PR_1, PR_2\}$.

It is expected that in rotating priority arbitration, different processors should get same chance for the shared resource i.e. processors have the same probability to occupy various positions within the priority queue. However, during experimentations, we observed that the processor with the highest request rate to the shared resource has maximum probability of getting the lowest priority. This happens because every time a request of this processor's request is served, the processor gets the lowest priority. We also observe that ordering of processors in the initial priority queue is important as it affects the probability of a processor having a particular priority.

At some instance, i^{th} processor with priority j can get the shared resource if no other processor with priority higher than it makes a request. There are a total of $Y = \binom{n-1}{j-1}$ possible combinations of processors having priority greater than j . If probability of y^{th} combination to appear is p_y and modified request rate of a processor PR_k in this combination is r'_k , then the probability b_{ij} that i^{th} processor with priority j successfully accesses the shared resource is:

$$b_{ij} = \sum^{Y \text{ terms}} p_y \times \left(\prod_{i \neq k}^{(j-1) \text{ terms}} (1 - r'_k) \right) \quad (3.6)$$

Now the acceptance probability of the processor i will be:

$$P_{a_i} = \sum_{j=1}^{j=n} p_{ij} \times b_{ij} \quad (3.7)$$

where p_{ij} is the probability that processor i has priority j . Accuracy of the estimation depends on choices of p_y and p_{ij} to a large extent. In this section, we present two different Models for these probabilities. Out of these, the first Model corresponds to the rotating priority arbitration policy of case 1 and second Model corresponds to case 2.

Model 1

We discussed in *Case 1* of rotating priority arbitration that the cyclic sequence of the processors in the priority queue is not maintained. Hence, in this case, it is assumed that it is equally likely that processor i has priority j . So, p_{ij} will be $\frac{1}{n}$. Under this assumption, equation 3.7 becomes:

$$P_{a_i} = \frac{\sum_{j=1}^{j=n} b_{ij}}{n} \quad (3.8)$$

Similarly probability for any processor combination p_y is assumed to be $\frac{1}{Y}$. In this case, ordering of processors in the priority queue is ignored. Now equation 3.6 becomes:

$$b_{ij} = \sum^{Y \text{ terms}} \frac{1}{Y} \times \left(\prod_{i \neq k}^{(j-1) \text{ terms}} (1 - r'_k) \right) \quad (3.9)$$

Model 2

We observed that every time the processor with the highest request rate is granted the shared resource, its priority becomes the lowest and hence if the priorities are rotated, then the processor next to it in the priority queue gets the highest priority. Essentially, the result is that the neighbor of the processor with the highest request rate gets the highest priority more frequently. This also makes p_{ij} unequal for different j^{th} priority positions. If number of processors is n , then index of the neighbor (processor PR_p) which gets i^{th} processor (PR_i) to j^{th} priority position will be $((i + n - j - 1) \bmod n)$. Since in *Model 2*, cyclic order of processors is always maintained, how frequently PR_i gets priority j is the function of modified request rate of PR_p . Hence, p_{ij} can be computed as follows.

$$\forall i \text{ and } j : \quad p_{ij} = \frac{r'_{(i+n-j-1) \bmod n}}{\sum_{k=1}^n r'_k} \quad (3.10)$$

In Case 2 of rotating priority arbitration implementations, cyclic order of processor sequence as specified in the initial priority queue is always maintained. Because of this, there is a possibility of only one processor combination to fill $(j - 1)$ higher priority slots when a processor is considered for priority j . Hence, equation 3.7 can be written as follows:

$$b_{ij} = \prod_{l=1}^{l=(j-1)} (1 - r'_{(n+i-j+1-l) \bmod n}) \quad (3.11)$$

Thus we get $2n$ simultaneous equations from 3.5 and 3.7. Values of p_{ij} and b_{ij} in equation 3.7 are obtained either from equations 3.9 and 3.8 or from equations 3.10 and 3.11 depending on the arbitration case. These can be solved iteratively for modified request rates of every processor. In experiments, we observed that the solution converges in a small number of iterations.

3.4.2 Multiple memory case

Figure 3.5 shows various states of processor i which makes request to two memories M_1 and M_2 . The processor remains in *Computing state* C_i unless there is a request to a memory connected to it. For each memory connected to this processor, there is one *Accepted state* A_{ij} and one *Waiting state* W_{ij} . i^{th} processor moves to the state A_{i1} when its access request made to the memory M_1 gets accepted. This transition can take place either from state C_i , from A_{i1} , or from A_{i2} . Otherwise processor goes to corresponding waiting state W_{i1} . Here, it keeps on resubmitting the request unless the request gets accepted. In Figure 3.5, r_{ij} is the request rate made to the memory M_j from i^{th} processor and Pa_{ij} is the probability that request made to the memory M_j from this processor will get accepted. State transition matrix for processor i can be derived from Figure 3.5 and corresponding state probabilities can be computed.

Let there be m memories and n processors. A processor makes a request with rate r_{ij} to the memory M_j from computing state C_i and all the accepted states A_{ij} . The processor always makes a request from waiting state W_{ij} . If q_{C_i} , $q_{A_{ij}}$ and $q_{W_{ij}}$ are the probabilities of being in states C_i , A_{ij} and W_{ij} respectively, then, modified request rates are obtained as follows.

$$\forall i \text{ and } j : \quad r'_{ij} = r_{ij} \times \left(q_{C_i} + \sum_{j=1}^{j=m} q_{A_{ij}} \right) + q_{W_{ij}} \quad (3.12)$$

Acceptance probabilities are computed in the same manner as for single memory case. Let p_{ijk} be the probability of processor i getting priority k at memory M_j . Similarly b_{ijk} is the acceptance probability for processor i at memory M_j when the processor has priority k .

$$\forall i \text{ and } j : \quad P_{a_{ij}} = \sum_{k=1}^{k=n} p_{ijk} \times b_{ijk} \quad (3.13)$$

From equations 3.5 and 3.13, we get $2 \times n \times m$ simultaneous equations which are solved iteratively. Next we describe the experimental results. These experiments were required to validate the estimation models discussed above against simulation results.

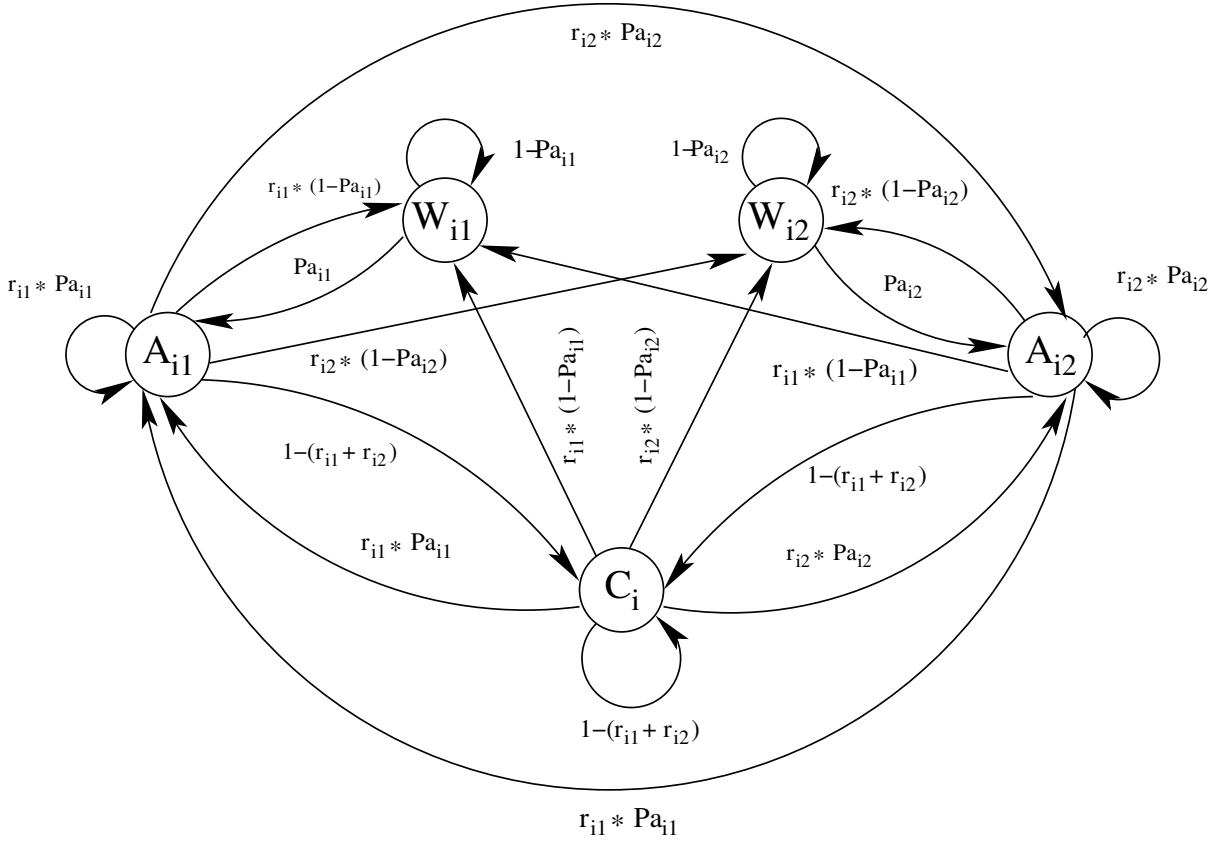


Figure 3.5: States of a processor making request to multiple memories

3.5 Experiments and Results

In heterogeneous multiprocessors, there are a large number of parameters which affect the performance. We considered the following parameters in our experiments:

1. *Number of processors and memories:* We varied number of processors in the range 3-7 and number of memories in the range 1-4.
2. *Processor to memory connectivities:* We specified random connectivity in processor-memory pairs. When single memory is present, all the processors communicate with it. However, in case of multiple memories, a processor could be connected to all the memories or to only some of them.
3. *Initial memory access request rates from processors to memories:* We specified initial

request rates (IRR) in the range 0.1-0.6 for each processor. IRR is nothing but the request rate of processor without re-submissions (r_{ij} : request rate from processor PR_i to memory M_j). Within a single experiment, request rates of the processors were specified in the range $Avg. IRR \pm \frac{n}{2} \times Delta$. Where n is the number of processors accessing a particular memory, $Avg. IRR$ is the average IRR across these n processors and $Delta$ is the minimum difference between the request rates of two processors accessing a single memory. For example, if 3 processors are connected to a single memory and their memory access rates are 0.3, 0.4 and 0.5 respectively, then $Avg. IRR$ is 0.4 and $Delta$ is 0.1.

The output of the estimation is the *modified request rate (MRR)* of each processor. We call it modified request rate because of initial request rate (IRR) of each processor directed to some memory gets modified due to re-submissions of request arising out of resource contention. MRR can be used to derive other performance parameters such as resource utilization etc. MRR is nothing but r'_{ij} as defined in the previous Section. In this Section, we are using symbols IRR and MRR instead of r_{ij} and r'_{ij} for the sake of simplicity.

We performed estimation of MRR of processors for the range of parameters specified above. This was done using both the Models discussed in Section 3.4.1. We implemented a probabilistic simulator in SystemC [73] to validate the estimation results. In this simulator, SystemC modules are connected as shown in Figure 3.1. Here processors randomly generate memory requests. These requests go through the interconnection network and arbitration is performed using the rotating priority schemes as discussed earlier.

In all the plots shown in this section, P_i is the i^{th} processor, E-MRR is the estimated modified request rate (MRR) and S-MRR is the simulated MRR. Figures 3.6(a) and 3.6(b) show MRR against Delta for different Avg. IRR when experiment was carried out on 4 processors connected to a single memory. In this experiment, simulation was done using arbitration case 1 and estimation was done using Model 1 discussed in Section 3.4.1. As expected, MRRs for different processors are same for homogeneous case (i.e. Delta is equal

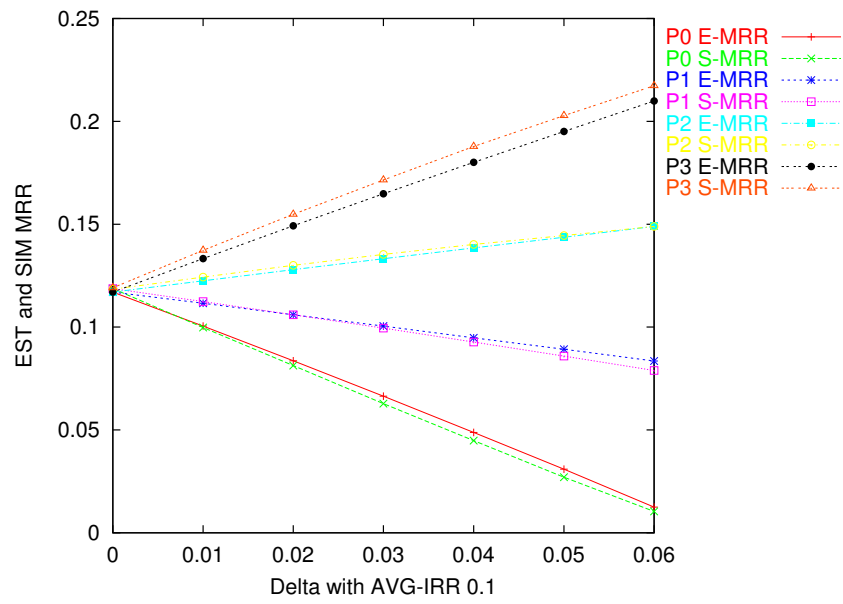
to 0) and increases as IRRs of the processors are increased. It can be noticed that the estimation closely follows the simulation.

Figures 3.7(a) and 3.7(b) show results when *Avg. IRR* was changed keeping *Delta* constant. In this case also, experimental setup was similar to the case discussed above. As in the above case, estimation results closely follow the simulation. Here we note that MRR seems to grow slower than linearly above $IRR \geq 0.4$ in Figure 3.7(b). In this case, MRR is certainly not going slower due to contention because in case of contention MRR is only expected to go up. It appears to be due to saturation. One reason of this saturation might be larger difference in IRR of the processors due to larger Delta (0.08 in this case).

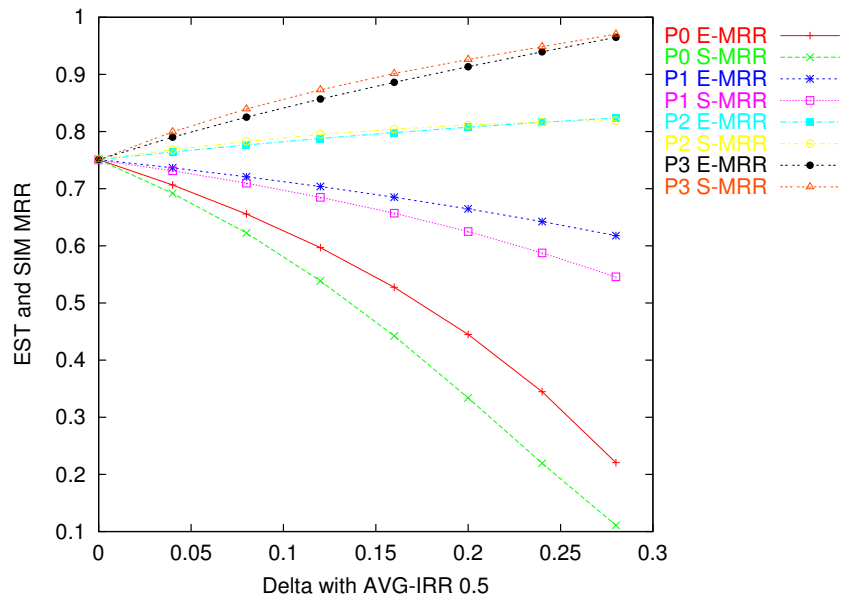
Figure 3.8(a) and 3.8(b) show results when 4 processors are connected to 3 memories. Memory 0 and 1 are accessed by all the processors, but Memory 2 is not accessed by processor P1. In this experiment, *Avg. IRR* to the memory 1 was similar to the memory 0, hence results are shown only for memory 0 and 2. Further, in simulation arbitration case 1 was used and estimation was done using Model 1. Here also, estimation results closely follow the simulation results.

In Section 3.4, we presented two Models corresponding to two different arbitration policy. Figure 3.9 shows the root mean square (RMS) error between MRRs of simulation and estimation when experiments were done with Avg. IRR of 0.3 and 6 processor to single memory connectivity. "*Arb. case i - Est. Model i*" in Figure 3.9 is the case when RMS is computed for the error between simulation results using arbitration case i and estimation using Model i, where i takes either the value 0 or 1. RMS errors are quite small in "*Arb. case 2 - Est. Model 2*". This is expected, since in this case, priority probabilities of different processors were modeled more accurately.

Error between estimation and simulation results for the range of experiments discussed earlier, are within 5-10% in most of the cases. The only exception is cases when request rates are higher (≥ 0.5). Furthermore, during estimations solution always converged, within 20 iterations.

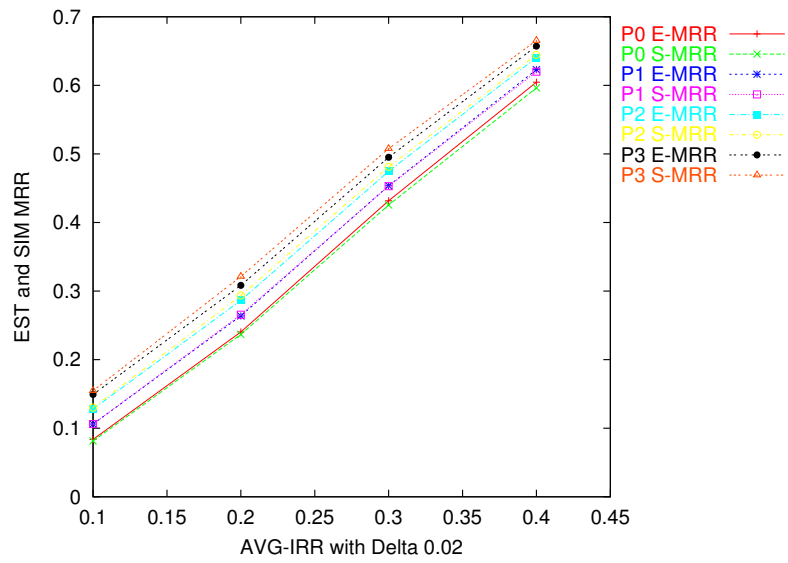


(a) Avg. IRR 0.1

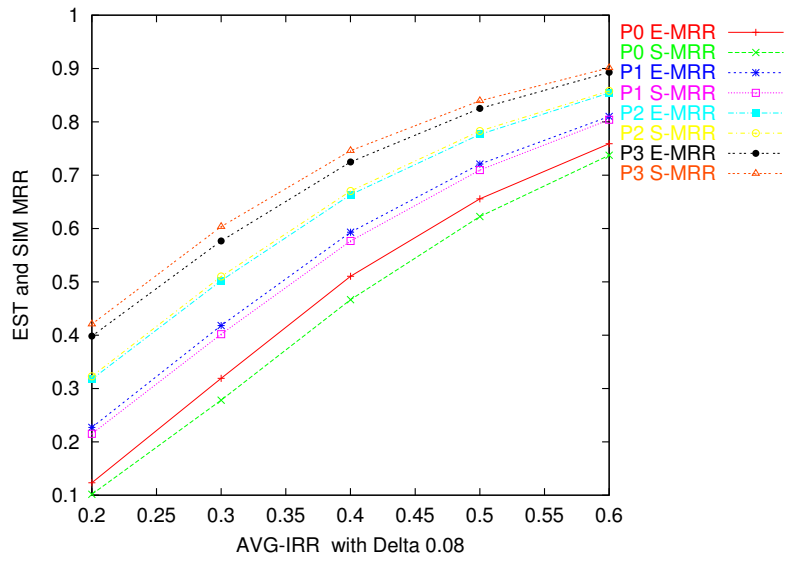


(b) Avg. IRR 0.5

Figure 3.6: MRR vs. Delta for single memory

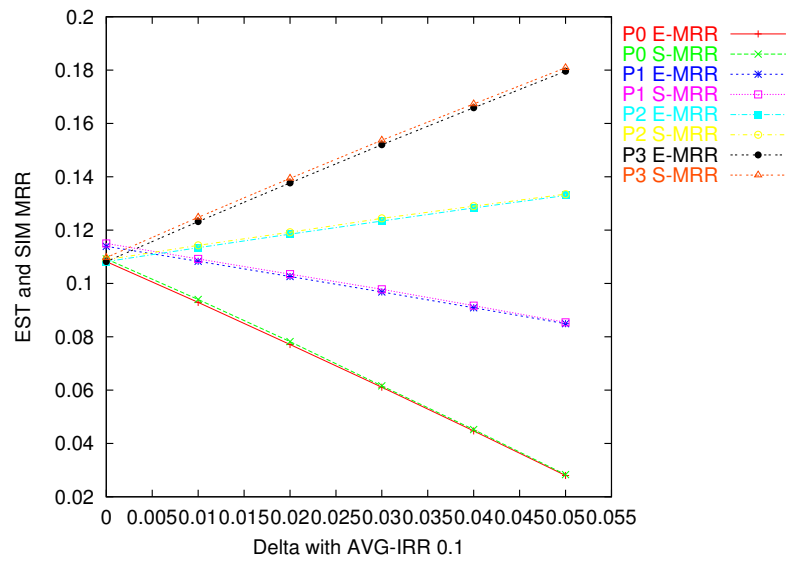


(a) Delta 0.02

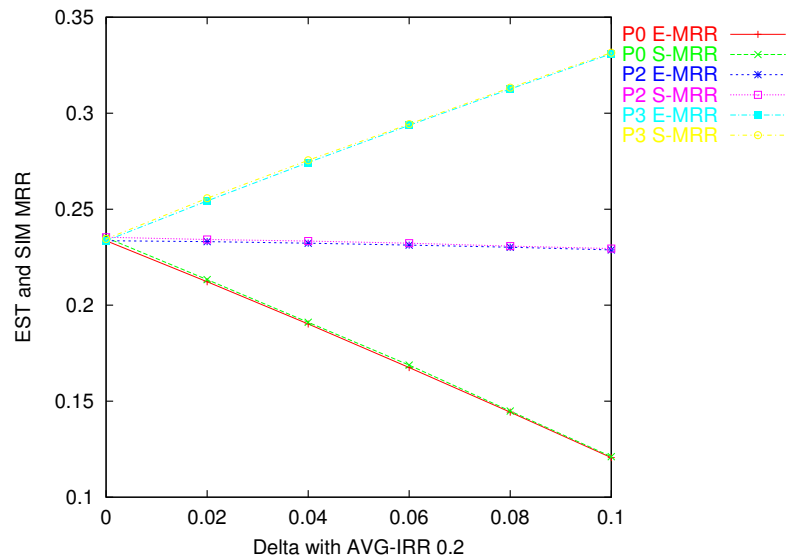


(b) Delta 0.08

Figure 3.7: MRR vs. Avg. IRR for single memory



(a) Avg. IRR 0.1 for Memory 0



(b) Avg. IRR 0.2 for Memory 2

Figure 3.8: MRR vs. Delta for multiple memory

3.6 Summary

We presented analytical approaches to estimate contention in shared memory heterogeneous multiprocessors having non-uniform memory access. Our method takes into account

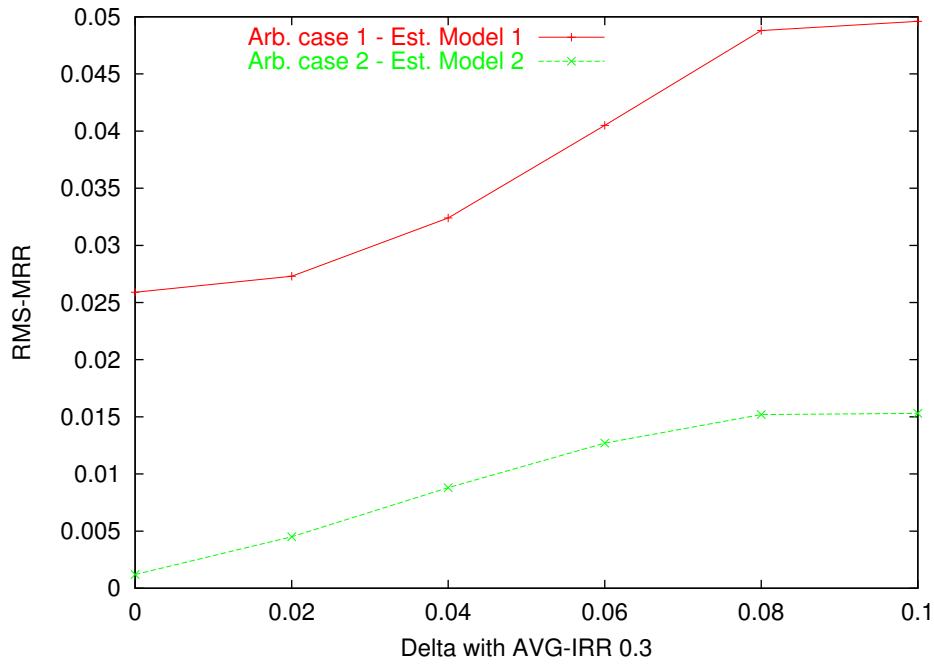


Figure 3.9: RMS error vs. Delta

the re-submissions of requests which are denied due to shared resource contention. We have modeled effect of arbitration policy more accurately. Our estimation results show very good correspondence to the simulation results. We have used these estimation approaches to evaluate a particular mapping of the application onto the heterogeneous multiprocessor architecture during design space exploration phase as discussed in Chapter 5

Chapter 4

MpSoC Synthesis Problem

Formulation

Given an application in the form of a KPN and a component library, synthesis of the architecture consists of the following to minimize total cost or energy.

1. Allocation of processors, local and shared memories
2. Binding of processes to processors and queues to local or shared memories
3. Allocation of communication components such as buses

Figure 2.6 shows an instance of a synthesized architecture. This example is repeated here in Figure 4.1 for convenience. In this example, the application KPN is composed of 3 processes and 3 queues. The synthesized architecture consists of 2 processors, 1 local memory and 1 shared memory. *Queue 1* is mapped to the local memory of *Processor 1* because reader and writer processes of *Queue 1* are mapped here. On the other hand *Queue 2* and *Queue 3* are mapped to the shared memory as their reader and writer processes are mapped to two different processors.

When the **objective function** of the synthesis problem is to minimize the hardware cost, then in the mapping stage we minimize the total cost of processors, local memory

modules, shared memory modules and interconnections cost. On the other hand, in case of energy minimization, we try to minimize energy consumed during execution of processes, memory accesses and data transfer on the interconnection network.

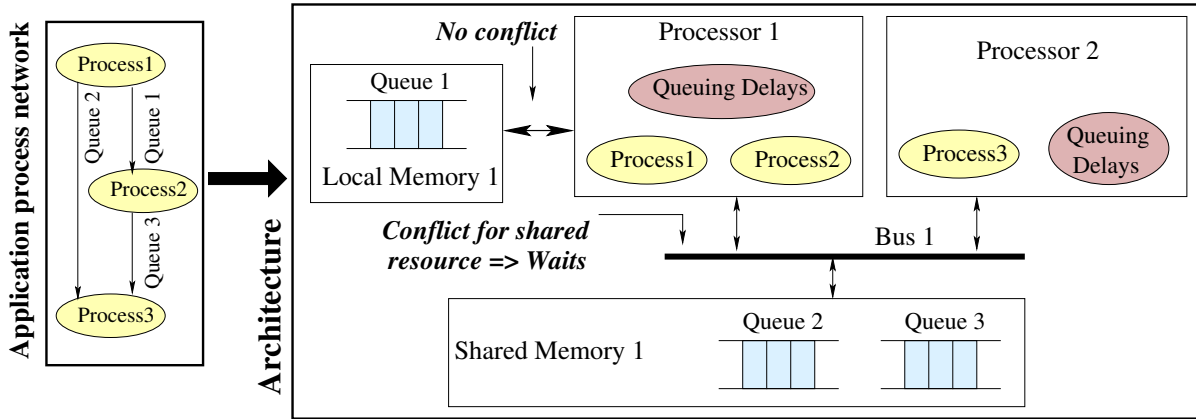


Figure 4.1: Synthesis example

In Section 4.3, we discuss the synthesis problem in detail and present a mixed integer linear programming (MILP) formulation for the resource allocation and application mapping. In this formulation, we assume that the component library already has sufficient number of various types of processors. However, in the heuristic based framework to solve the synthesis problem described in Section 5.1, we relax this restriction and instantiate a particular type of processor present in the component library only when required. Section 4.4 describes the MILP for the communication architecture followed by the summary of the Chapter.

4.1 Application Parameters

As discussed in Section 2.1.2, we consider applications modeled as KPN. We assume that the processes are iterative in nature and perform computation as soon as required data is available at their inputs. This is a reasonable assumption for the streaming applications. The KPN model can have more than one channel between two processes. It can also

have cycles. Unlike begin-end type of task graphs, here computation and communication are interleaved. Hence in our application model, an arc only means that there is some communication from one process to another during the course of computation. Various process network parameters are given in Table 4.1.

T_i	i^{th} process
Q_j	j^{th} channel
thr_j	throughput constraint in terms of number of tokens to be produced/consumed per second on Q_j
$iter_i$	iterations/second must be made by T_i
$qwr(j)$	index of the writer process of channel Q_j
$qrd(j)$	index of the reader process of channel Q_j
$T_{qwr(j)}$	writer process of channel Q_j
$T_{qrd(j)}$	reader process of channel Q_j
sz_j	size in bytes of the token of Q_j
len_j	length of Q_j
$ntkn_{ij}$	number of tokens produced/consumed per iteration by T_i on/from Q_j

Table 4.1: Notations for application parameters

Real time constraint is specified in terms of how often data tokens should be produced on the channels and consumed from the channels. We represent this as throughput constraints on channels of the process network. thr_j the is throughput constraint in terms of number of tokens to be produced/consumed per second for queue Q_j . sz_j is the size in bytes of a token of queue Q_j . Here we assume that the size of a token produced or consumed on a queue is same.

$ntkn_{ij}$ is the number of tokens produced or consumed by T_i on/from Q_j in one iteration of T_i . This number is more than 0 if T_i is reader or writer of Q_j otherwise it is 0. Throughput constraints of channels are used to derive the constraints on the processing requirements of processes. $iter_i$ is the constraint on a process such that it should make at least $iter_i$

iterations/sec. This is derived from the throughput constraints on the queues and equals to the maximum of $\frac{thr_j}{ntkn_{ij}}$ over all the queues connected to the process T_i .

4.2 Architectural Parameters

We assume an architecture component library which contains a number of processors, memory modules and interconnection components. Various architecture dependent parameters are given in Table 4.2. The k^{th} processor is denoted as PR_k . pr_cost_k is the cost of processor PR_k . V_k is the set of operating voltage levels of PR_k . This set contains a single value if the processor does not have dynamic voltage scaling feature. clk_{vk} is the clock period of PR_k at voltage v such that $v \in V_k$. The maximum frequency of processor PR_k is $maxfreq_k$. Number of cycles taken by a process during a context switch at processor PR_k is denoted by $contxt_k$.

ncy_{ik} is the number of cycles taken by the process T_i when mapped onto processor PR_k for one iteration without memory conflicts and context switch overheads. This can be either obtained using estimations or using simulations. In case, it is not feasible to map T_i on PR_k , we assign ncy_{ik} a very large value. en_{ivk} is the energy consumed by the process T_i when mapped onto processor PR_k for one iteration at clock period clk_{vk} .

Alongwith a processor, there is a local memory. This local memory contains channels which have their reader and writer processes mapped onto the corresponding processor. Local memory also contain the local variables which hold the value of a single token of a particular channel. Processors are allowed to write/read value of token of any channel only through local memory. This ensures that there is no data consistency problem due to sharing of data at shared memories. This can be easily seen from the fact that local variable corresponding to a token of the channel is either written or read only once for each individual token.

There are also a number of shared memory modules shared among processors. k^{th}

PR_k	k^{th} compute unit
pr_cost_k	cost of PR_k
$maxfreq_k, minclk_k$	maximum frequency of PR_k , minimum clock period of PR_k
V_k	set of operating voltage levels of PR_k
clk_{vk}	clock period of PR_k at voltage $v \in V_k$
$contxt_k$	number of cycles consumed during a context switch on PR_k
ncy_{ik}	number of cycles taken by the process T_i when mapped onto processor PR_k for one iteration without memory conflicts and context switch overheads
en_{ivk}	energy consumed by the process T_i when mapped onto processor PR_k for one iteration at clock period clk_{vk}
LM_k, SM_l	local memory alongwith PR_k , l^{th} shared memory
$cost_base_lm_k$	base cost of LM_k
$cost_base_sm_l$	base cost of SM_l
$bwmm_l$	bandwidth of SM_l in bytes/sec
SW_m	m^{th} switch
M_m, N_m	number of processor side ports, number of memory side ports
$swty_m$	type of switch SW_m . Value 1 denotes a bus and 0 denotes cross-bar switch
$cost_sw_m$	overall cost of SW_m
$cost_sw_in$	cost of a link associated
bws_w_m	bandwidth of SW_m

Table 4.2: Notations for architecture parameters

local memory is represented as LM_k and l^{th} shared memory module is denoted as SM_l . $cost_base_sm_l$ and $cost_base_lm_k$ are the base cost of SM_l and LM_k respectively. The cost depends on its size. $bwmm_l$ is the bandwidth of SM_l in bytes/sec.

Interconnection components consist of a number of switches. m^{th} switch is denoted by SW_m . Switch SW_m has M_m number of processor side ports and N_m number of memory side ports. Every switch is assigned a type $swty_m$. Value 1 denotes that switch is a bus and

0 denotes that it is a cross-bar switch. Here buses are represented as low cost cross-bar switches providing full connectivity. A switch has two costs associated with it: $cost_sw_m$ which is the overall switch cost and $cost_sw_in_m$ which is the cost of associated links. The former cost is equal to $C \times M_m \times N_m$ for cross-bar switches. Here C is some switch implementation dependent constant. Latter cost is introduced to take into account the case of multiple buses. The main difference between single bus and multiple bus is that latter has higher bandwidth and a component attached to it will have as many connections to it as many buses in it. $cost_sw_in_m$ captures this higher link cost. We also associate bandwidth parameter bws_w_m with every switch to take into account the bandwidth limitations of buses.

4.3 MILP for Computing Resources and Application Mapping

4.3.1 Decision Variables

Basic Mapping Variables

These are the binary variables which define the architecture instance and mapping of application onto the architecture. tp_{ip} becomes 1 when process T_i is mapped to processor PR_p . qlm_{jp} is 1 if queue Q_j is mapped to local memory LM_p . Similarly, qmm_{jl} becomes 1 if queue Q_j is mapped to shared memory module SM_l .

Derived Variables

These binary variables are derived from the basic decision variables. pmm_{pl} becomes 1 if processor PR_p reads from or writes into the shared memory module SM_l . pmm_{pl} is defined by Equation 4.5. $QQmm_{rs}$ is 1 if queues Q_r and Q_s are mapped to the same shared memory module. $QQmm_{rs}$ has been defined using Equation 4.6. Similarly, upr_p becomes 1 when

processor PR_p is being used. These variables are obtained by Equation 4.7.

4.3.2 Mapping Constraints

These are the constraints which define mapping of process network and architecture instance.

A particular process T_i can be mapped onto only one processor.

$$\forall i : \quad \sum_p tp_{ip} = 1 \quad (4.1)$$

A queue Q_j is mapped onto a local memory only when its reader and writer processes are mapped to the same processor.

$$\forall p, j : \quad qlm_{jp} = tp_{qwr(j),p} \wedge tp_{qrd(j),p} \quad (4.2)$$

In above equation, $qwr(j)$ and $qrd(j)$ are the indices of writer and reader processes of queue Q_j respectively. This equation is non-linear and can be linearized as follows.

$$\forall p, j : \quad qlm_{jp} \geq tp_{qwr(j),p} + tp_{qrd(j),p} - 1 \quad (4.3)$$

Right hand side in equation 4.3 will become 1 as soon as the reader and writer processes are mapped to a single processor.

A queue is mapped to either local memory of a processor PR_p or shared memory module SM_l .

$$\forall j : \quad \sum_p qlm_{jp} + \sum_l qmm_{jl} = 1 \quad (4.4)$$

A processor PR_p will communicate with a memory module SM_l when some reader or writer of a queue Q_j is mapped onto PR_p and queue itself is mapped onto SM_l .

$$\forall p \text{ and } l : \quad pmm_{pl} = \bigvee_j qmm_{jl} \wedge (tp_{qwr(j),p} \vee tp_{qrd(j),p}) \quad (4.5)$$

$QQmm_{rs}$ is 1 only when both the queues Q_r and Q_s are mapped to the same shared memory module SM_l .

$$\forall r \text{ and } s : \quad QQmm_{rs} = \bigvee_l qmm_{rl} \wedge qmm_{sl} \quad (4.6)$$

We make the equations 4.5 and 4.6 linear by using a similar approach as followed to make the equation 4.2 linear.

A processor PR_p is utilized only if some process T_i is mapped onto it.

$$\forall p \text{ and } i : \quad upr_p \geq tp_{ip} \quad (4.7)$$

4.3.3 Performance Constraints

Capacity Constraint at Processors

Referring to Figure 4.1, we note that there are three possible states of an instantiated processor. Either processor is executing some process, or it is switching context from one process to another or it is waiting due to conflicts while accessing shared memories. A processor executes the number of instructions equal to its frequency in one second. We call it processor's capacity. Now, total number of cycles required, considering all the possible states of the processor in one second, must not exceed its capacity. This is what we check while verifying whether performance constraints at a particular processor is satisfied or not.

Context Switch Overheads

When multiple processes are mapped onto a single processor, there will be a context switch overhead. A process gets blocked while reading a token from an empty queue or when writing a token into queue which is full. We notice that a writer process gets blocked only when it has written at least len_j (maximum number of tokens in Q_j) tokens on Q_j . Similarly, a reader process gets blocked only when it has read all the tokens placed in that

queue. Here, we assume that a process can be blocked only when it is communicating. Hence, the number of times the queue Q_j will get filled/emptied will be $\frac{thr_j}{len_j}$. This is also the number of times the writer process ($T_{qwr(j)}$) and the reader process ($T_{qrd(j)}$) will get blocked. So, if the set of queues which have either their reader or writer mapped onto a processor PR_p is QPR_p , then the number of context switches per second ($ncontxt_p$) on PR_p can be computed as follows.

$$\forall p : \quad ncontxt_p = \sum_{j : Q_j \in QPR_p} \frac{thr_j}{len_j} \times (tp_{qwr(j),p} + tp_{qrd(j),p}) \quad (4.8)$$

Queuing Delays

The mutual interference will appear during accesses on two queues only when they are mapped to the same shared memory module and respective readers and writers are mapped to different processors. We want to find out the equivalent waiting time for the processor. Consider the synthesis example shown in Figure 4.1. If somehow, we get the queuing delay processor 1 will face when it makes a read/write access on Queue 2, we can compute the total overhead per second. To compute it, queuing delays are multiplied by the throughput requirements of Queue 2.

In Section 3.3, we described an approach to estimate the queuing delays faced by individual processors while accessing some queue on per access basis. We used this method and pre-computed all possible kinds of queuing delays (read-read, write-write, read-write and write-read). We do this for all the processor pairs present in the component library and each pair of queues. Here write-read refers to the case when processor 1 is making a write access to one of the queues and processor 2 is making a read access simultaneously as in Figure 4.1.

Let us consider the case when, two queues Q_r and Q_s are mapped to the same shared memory instance and their readers are mapped to different processor instances PR_p and PR_q (term $(QQmm_{rs} \wedge tp_{qrd(r),p} \wedge tp_{qrd(s),q})$ becomes 1). Then, as defined in Section 3.3,

the equivalent waiting delay for each read request on Q_r from PR_p is $rrcon_{rspq}$. Hence, the total read-read queuing delay as per throughput constraint for this mapping can be computed as follows.

$$data_rrcon_{rsp} = rrcon_{rspq} \times thr_r \times (QQmm_{rs} \wedge tp_{qrd(r),p} \wedge tp_{qrd(s),q})$$

After making it linear, it becomes:

$$\begin{aligned} \forall r, s, p \text{ and } q: \quad data_rrcon_{rsp} \geq & rrcon_{rspq} \times thr_r \times \\ & \times (QQmm_{rs} + tp_{qrd(r),p} + tp_{qrd(s),q} - 2) \end{aligned} \quad (4.9)$$

Similar equations can also be written for write-write ($data_wwcon_{rsp}$) cases.

Capacity Constraint

As we discussed earlier, a processor offers capacity equal to its clock frequency (cycles per second). This must accommodate computation requirements of processes mapped, context switch overheads and waiting time due to data communication queuing delays. If the process T_i mapped to processor PR_p , executes at clock period clk_{vp} , then total time taken by this process will be $ncy_{ip} \times iter_i \times clk_{vp}$. Assuming that PPR_p is the set of processes mapped to processor instance PR_p , then total computation requirements of the processes will be $\sum_{i \in PPR_p} (ncy_{ip} \times iter_i \times clk_{vp})$. Equation 4.8 gives us the number of context switches on processor PR_p . Hence, total context switch delay on this processor becomes $contxt_p \times ncontxt_p \times minclk_k$. The last component of capacity constraint is the queuing delays. These are computed for a queue using Equation 4.9. So, the total waiting time due to queuing delays becomes $\sum_r \sum_s (data_rrcon_{rsp} + data_wwcon_{rsp})$. Now, the capacity constraint can be written as follows.

$$\begin{aligned} \forall p: \quad \sum_{i \in PPR_p} (ncy_{ip} \times iter_i \times clk_{vp}) + (contxt_p \times ncontxt_p \times minclk_k) + \\ + \sum_r \sum_s (data_rrcon_{rsp} + data_wwcon_{rsp}) \times minclk_k \leq 1 \end{aligned} \quad (4.10)$$

If the processors don't have the dynamic voltage scaling feature, then above equation

reduces to the following.

$$\begin{aligned} \forall p : \sum_{i \in PPR_p} (ncy_{ip} \times iter_i) + contxt_p \times ncontxt_p + \\ + \sum_r \sum_s (data_rrcon_{rsp} + data_wwcon_{rsp}) \leq maxfreq_p \end{aligned} \quad (4.11)$$

Bandwidth Constraint at Shared Memories

Arrival rate of read or write requests at a shared memory module SM_l , depends on communication rates and sizes of data transfer during each request. This request is at some queue of the process network mapped onto SM_l . Bandwidth $bwmm_l$ of this shared memory module should be larger than the total arrival rate of all requests. If thr_j is the throughput constraint for queue Q_j mapped to shared memory SM_l and sz_j is token size for it, then

$$\forall l : \sum_j qmm_{jl} \times thr_j \times sz_j \leq bwmm_l \quad (4.12)$$

We discussed above that waiting delays due to conflicts at shared memories are taken into account in the capacity constraint of the processor. Hence, the memory bandwidth constraint given by Equation 4.12, is to check whether maximum request rate at a particular shared memory can be sustained.

Objective function

The *objective function* in the synthesis problem is to minimize hardware cost. In the mapping stage, it essentially consists of cost of resources i.e. processors used, local memory modules, shared memory modules and interconnection cost.

$$minimize : processor_costs + LM_costs + SM_costs + interconnect_cost$$

Cost of a processor depends on its type. If cost of processor PR_p is pr_cost_p , then total cost of utilized processors becomes:

$$compute_units_costs = \sum_{PR_p} pr_cost_p \times upr_p$$

Cost of a shared memory module depends on how much space is being occupied by the queues of the process network mapped onto it. This in turn depends on sizes, lengths of the queues and equivalent hardware cost of mapping a single word ($cost_base_sm_l$) onto the shared memory module SM_l .

$$SM_costs = \sum_l \sum_j (qmm_{jl} \times cost_base_sm_l \times sz_j \times len_j)$$

Cost of the local memory module is determined in the same manner as that of a shared memory module.

$$LM_costs = \sum_p \sum_j (qlm_{jp} \times cost_base_lm_p \times sz_j \times len_j)$$

Interconnect cost depends on the processor memory connectivity. So, total interconnect cost can be computed as follows.

$$interconnect_cost = \left(\sum_p \sum_l pmm_{pl} \right) \times cost_sw_in$$

4.4 MILP for Communication Architecture

Synthesis of communication architecture can either be done alongwith the previous stage (mapping) or as a post processing step when mapping is already known. Former leads to overall minimum cost solution. Latter significantly simplifies the MILP of mapping stage, but this might lead to overall higher cost solution.

Decision Variables

Binary variables ps_{km} and smm_{ml} define connectivity of the components. ps_{km} becomes 1 if PR_k is connected to switch SW_m . Similarly, smm_{ml} is 1 when SM_l is connected to switch SW_m . Binary variables $psmm_{kml}$ and usw_m are derived from the basic variables. $psmm_{kml}$ becomes 1 if there is a path $PR_k - SW_m - SM_l$ and usw_m becomes 1 if SW_m is being used.

Constraints

Like shared memories, buses are also shared resources. We compute total bandwidth requirement because of accesses being made on the shared memory modules.

$$bw_req = \sum_j \sum_l qmm_{jl} \times thr_j \times sz_j$$

A switch of type bus cannot be used if it does not meet bandwidth requirement.

$$usw_m = 0 \quad \text{if } swty \text{ is } 1 \text{ and } bws_w_m \leq bw_req$$

Number of compute units connected to a switch should not be greater than number of processor side ports of the switch. Similarly number of memory modules connected to a switch should not be greater than number of memory side ports of a switch.

$$\begin{aligned} \forall m : \quad \sum_k ps_{km} &\leq M_m \\ \forall m : \quad \sum_l smm_{ml} &\leq N_m \end{aligned} \quad (4.13)$$

There is one path from PR_k to SM_l if they communicate with each other. This is required to reduce interconnections.

$$\forall k, \text{ and } l : \quad \sum_m psmm_{kml} = pmm_{kl} \quad (4.14)$$

Compute unit PR_k is connected to switch SW_m if there is at least one communication of PR_k to some memory module SM_l . Similarly, memory module SM_l is connected to switch SW_m if there is at least one communication of SM_l to some compute unit PR_k .

$$\begin{aligned} \forall k, \text{ and } m : \quad ps_{km} &\geq \bigvee_l psmm_{kml} \\ \forall m, \text{ and } l : \quad smm_{ml} &\geq \bigvee_k psmm_{kml} \end{aligned} \quad (4.15)$$

Above equation can be easily linearized as earlier done. Now, a switch is utilized only if some compute units and some memory module are connected to it.

$$\begin{aligned} \forall m, \text{ and } k : \quad usw_m &\geq ps_{km} \\ \forall m, \text{ and } l : \quad usw_m &\geq smm_{ml} \end{aligned} \quad (4.16)$$

Objective function

The objective function in communication architecture synthesis is to minimize the total cost of switches being used and associated interconnections.

$$\text{minimize : } \text{switches_costs} + \text{interconnection_cost}$$

$$\text{switches_costs} = \sum_m \text{cost_sw}_m \times \text{usw}_m$$

Cost of the interconnects linearly depends on number of connections from compute units to switches and switches to memory.

$$\text{interconnection_cost} = \sum_m \left(\sum_k p_{skm} + \sum_l \text{sml}_{ml} \right) \times \text{cost_sw_in}$$

4.5 Summary

In this Chapter, we described the MpSoC synthesis problem. We formulated MpSoC synthesis as two mixed integer linear programming problems. The first MILP can be used for allocation of processors, memories and mapping of application onto the architecture. The second MILP can be used to explore various interconnection networks. These two MILPs can be given to the solver together which leads to the overall minimum cost. These MILPs can also be solved separately which may not give the overall minimum cost solution, but significantly simplifies the problem complexity. The MILP solver takes very large amount time for larger problem instances. So we worked out a heuristic based framework for the MpSoC synthesis problem. We describe this framework in the next Chapter.

Chapter 5

MpSoC Synthesis Heuristics

In Chapter 4, we described the MILP formulation for the MpSoC synthesis problem. The MILP solver takes very large amount time for larger problem instances. So, for this problem, we developed a heuristic based framework which is described in this Chapter. Our synthesis framework solves the exact problem as described in Chapter 4, but it treats the architectural component library slightly differently. The differences in the treatment are discussed next.

In the MILP formulation, we didn't have the notion of processor types in the component library. It was assumed that a number of processors are present in the library and some of the processors could be of the same type, but they needed to be considered separately. So, if there are only 3 processors of the same type in the library, the MILP solver cannot instantiate more than 3 processors of the same type. This was done to avoid explosion of number of decision variables. Shared memory modules were treated in the similar manner. In the MpSoC synthesis framework, described in this Chapter, we relax the above restriction. In this case, we specify the component library in terms of processor and memory types. Further, there is no restriction on the number of instances of a processor or memory type and a number of instances of these modules can be created depending on the requirements.

So as discussed above, the difference between problems solved using MILP solver and the ones using heuristic approach lies in the treatment of architecture library. Both the problems become same when we put an upper bound on number of processors of each type which can be instantiated within the synthesized architecture using heuristic approach and keep the number of processors equal to the upper bound for each type in the architecture library for MILP solver case. Same thing needs to be done for the shared memory modules as well.

The MpSoC synthesis framework discussed in this Chapter, works in two stages. In the first stage, we get the initial solution by a constructive algorithm. This is described in Section 5.1. In the second stage, iterative refinements of the initial solution generated in the first stage are done. Section 5.2 describes this stage. The power optimization is discussed separately in Section 5.3. This optimization can be called within both the stages mentioned above. Summary of the Chapter follows in Section 5.5.

5.1 Construction of Initial Solution

We propose a heuristic based solution for our synthesis problem. We construct the solution by employing a fast dynamic programming based algorithm. Our approach not only allows it to be used in a design space exploration loop, but also provides a good initial solution for an iterative refinement phase.

We note that the following considerations help us to minimize the cost of synthesized architecture.

- Mapping a process onto a low cost processor.
- Mapping densely connected processes to the same processor will result in putting more queues in the local memory. This in turn helps to reduce interconnection network (IN) cost.

- Mapping queues, where communication takes place more frequently, onto the local memories will help to reduce conflicts at shared memories.

One can make the observation that out of above considerations, last two become effective when two adjacent processes from the process network graph are mapped to the same processor. The condition is that these two processes should either be communicating over a number of queues or communicating more frequently. So, mapping such adjacent processes onto lower cost processors would result in a cost effective solution. This is the basis of our synthesis Algorithm to create the initial solution, where we first create a list of adjacent processes and then try to map them onto one processor wherever possible.

In the synthesis framework, the heuristic is being used to generate sorted list of processes which is further given to the dynamic programming algorithm to provide the solution. We considered some heuristics including assigning all the strongly connected components of the process network onto a single processor. There are two problems with this heuristic.

1. It may not be feasible to assign all the processes of a strongly connected component of the process network as there might not be a processor in the architecture library satisfying performance requirement.
2. We might miss assigning two processes which are heavily communicating with each other onto a single processor.

Because of the above reasons, it makes more sense to create an adjacent list of processes in which two adjacent processes heavily communicate with each other. There are the following advantages of this approach.

1. Incremental synthesis can be performed by mapping partial list and hence arriving at a feasible solution.
2. High priority can be given to the local communication.

The synthesis framework described next doesn't impose an upper bound on the number of processors and memories instantiated. So, for each process as long as there is at least one processor which can handle its computation requirement and for each channel there is at least one memory type capable of handling its communication requirements, we will get a feasible solution for the given process network.

Before describing the Algorithm, we will first define the partial map and then, discuss how we get the adjacent processes.

5.1.1 Definition of Partial Map

We define the partial map (PM) for a set of processes. PM for this set essentially contains its mapping information i.e. binding information of processes of this set to processors and queues (having their reader and writer processes in this set) to memories. For example, for a set of processes $[T_4, T_3, T_5, T_0]$ of Figure 5.1, the partial map is shown Figure 5.2. We note that this set is mapped onto two processors. We term this as multiple processor mapping (MPM). Furthermore, partial map for $[T_4, T_3]$ contains a single processor and associated local memory. We term this as single processor mapping (SPM). Partial map of $[T_5, T_0]$ has similar structure. We also note that queues other than what has been shown in Figure 5.2 are still unassigned to any memory. That is why this mapping information is partial.

5.1.2 Creating Adjacent Process List

To create a list of adjacent processes, we derive undirected CRG from the original process network in step 1 of Algorithm 1. CRG is derived by collapsing all the edges between two vertices in original process network into a single edge annotated with total communication between them. Hence, weight on an edge between two processes T_a and T_b in CRG becomes $\sum_{Q_j} (sz_j \times thr_j)$. Here, Q_j is any queue between these processes, sz_j is the size of a token being transferred over this queue and thr_j is its throughput constraint.

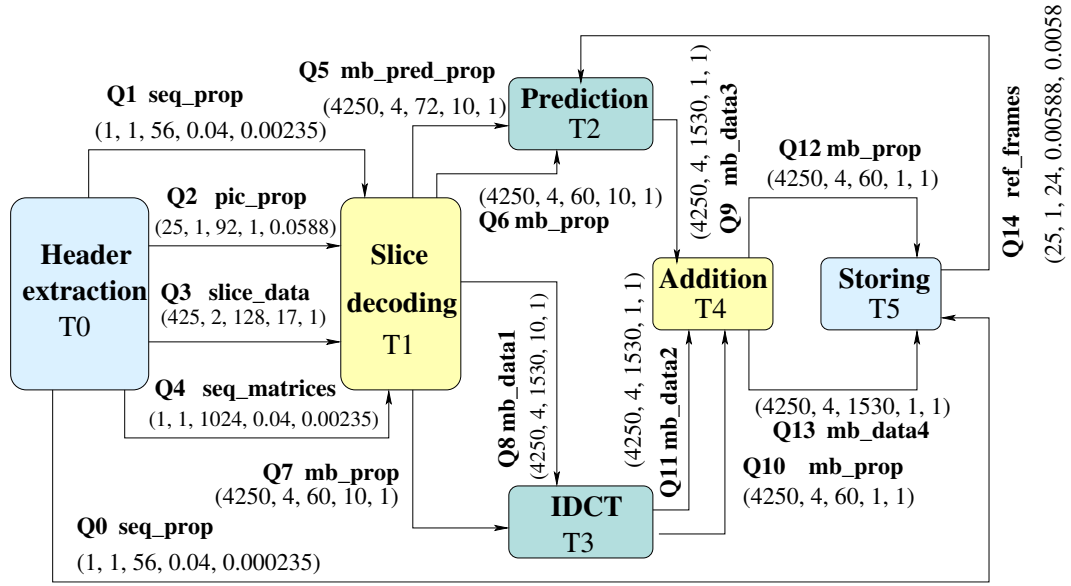


Figure 5.1: Annotated MPEG-2 video decoder process network

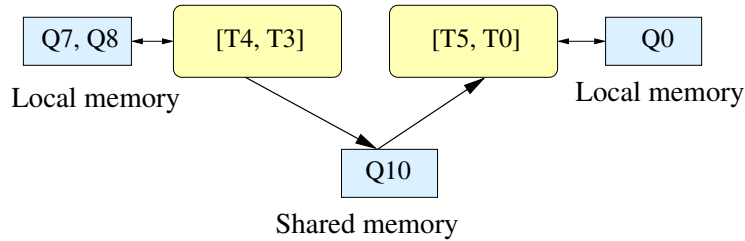


Figure 5.2: Example partial map

Figure 5.1 shows annotated MPEG-2 video decoder. Here 5-tuple next to each edge is the parameter values for size of a token being transferred, the maximum number of tokens allowed in the queue, throughput constraint, number of tokens produced or consumed by the writer or reader process in its single iteration respectively. Corresponding CRG is shown in Figure 5.3.

Next, the vertex, having maximum weight on one of its edges in CRG, is chosen as *root*. Weighted topological sort is performed on CRG starting from this vertex. In this phase,

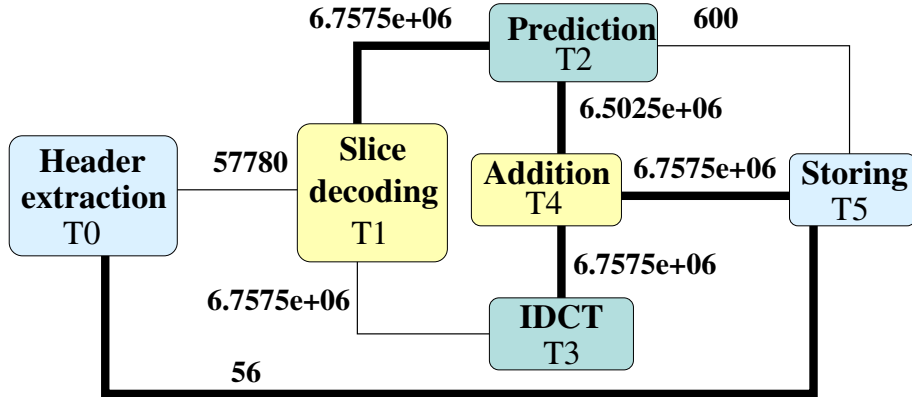


Figure 5.3: CRG of MPEG-2 video decoder

a new edge on the path is chosen which has the maximum weight and leads to another vertex which has not yet been visited. In Figure 5.3, it starts from T_1 and takes the path $T_1 \rightarrow T_2 \rightarrow T_4 \rightarrow T_3$. The whole path is deleted from CRG and new *root* is chosen if there is no path from the last node on this path to other vertices. For this example, sequence $\{T_1, T_2, T_4, T_3, T_5, T_0\}$ is the *adjacent_process_list*. From now onwards, we will use term partial map (*PM*) only for processes ordered in *adjacent_process_list*.

5.1.3 Computation of Partial Map

Processes of *adjacent_process_list* can be mapped onto different processors in a number of ways. Figure 5.4 shows two such possibilities for the process network of Figure 5.1. Next, we assume that there are processors of three types PR_1 , PR_2 and PR_3 in the component library. If we cannot map the whole *adjacent_process_list* onto a single processor due to performance constraints, then we need to break this list into two and evaluate them separately. This can be done in a number of ways which correspond to various sub-trees in Figure 5.4. We see that in sub-tree 1, we can map process group $[T_1, T_2]$ onto processor PR_1 , but process group $[T_4, T_3, T_5, T_0]$ couldn't be mapped onto a single processor (perhaps because of violation of performance constraints). Hence, this list needs to be broken further. We stop exploring this sub-tree as soon as all the group of processes in it are mapped onto

some processor under performance constraint. Similarly other sub-trees are also evaluated and the lowest cost sub-tree is selected as the solution.

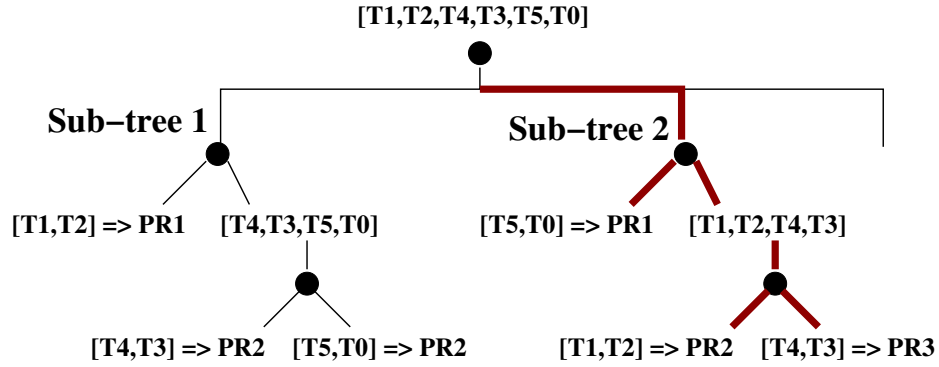


Figure 5.4: Example solution tree

We observe that the architecture can be synthesized by building partial maps recursively in a bottom up manner. This suggests that a dynamic programming based algorithm can be used for synthesis. In step 3 of Algorithm 1, partial map matrix PM for the solution is created using a dynamic programming algorithm (Equation 5.3). Here, partial map $PM[m, n]$ refers to the mapping information of processes in *adjacent_process_list* between indices m and n .

Algorithm 1 *synthesize_architecture*

- 1: Derive Communication Requirement Graph (CRG)
 - 2: Create *adjacent_process_list* by performing weighted topological sort on CRG starting from its *root* vertex
 - 3: Compute partial map $PM[m, n]$ for processes for each $m \rightarrow n$ such that $m \leq n$ in *adjacent_process_list*
 - 4: Refine architecture using shared memory merging
-

In the first three steps of Algorithm 1, we incrementally build partial map for all the processes. Now, there exists two possible solutions for the partial mapping of processes of

adjacent_process_list.

Solution 1: Single processor mapping

First we evaluate whether all processes in *adjacent_process_list* between indices m and n can be mapped onto the same processor from the component library. We choose the lowest cost processor PR_k which satisfies performance constraints. In this step, performance constraint is evaluated using Equations 4.10 and 4.11. If there is no such processor, then cost of this solution is infinite. Otherwise, single processor partial mapping ($SPM[m, n, PR_k]$) cost becomes:

$$SPM[m, n, PR_k].cost = pr_cost_k + \sum_{Q_j} sz_j \times len_j \times cost_base_lm_k \quad (5.1)$$

where reader and writer processes of the queue Q_j are in group $m \rightarrow n$ itself. In Equation 5.1, the cost is composed of cost of the processor chosen (pr_cost_k) and cost of the local memory. Latter is equal to the equivalent hardware cost of mapping single word into the local memory ($cost_base_lm_k$) multiplied by the total local memory required ($\sum_{Q_j} sz_j \times len_j$). For example, if we are computing partial map $PM[2, 3]$ ($[T_4, T_3]$) for *adjacent_process_list* of Figure 5.3 and these processes can be mapped onto a single processor, then queues Q_{10} and Q_{11} will be mapped to the local memory. Rest of the queues connected to this partial map will remain unassigned.

Solution 2: Multiple processor mapping

Next, a pivot p in *adjacent_process_list* is moved from m to n . If reader and writer of queue Q_j are in groups $m \rightarrow p$ and $(p + 1) \rightarrow n$ or vice versa, then cost of multiple processor partial map ($MPM[m, p, n]$) is composed of 4 components: cost of partial maps $PM[m, p]$ and $PM[p + 1, n]$, cost of new shared memories instantiated and cost of new links introduced. To illustrate, let us consider *adjacent_process_list* of Figure 5.1 again.

While combining partial maps $PM[2, 3]$ ($[T_4, T_3]$) and $PM[4, 5]$ ($[T_5, T_0]$), we instantiate new shared memories for queues Q_{12} and Q_{13} because partial maps communicate via these queues. As shown in Figure 5.2, two new links are also introduced for each queue. Hence, total cost of new partial map is computed as follows.

$$\begin{aligned} MPM[m, p, n].cost &= PM[m, p].cost + PM[p + 1, n].cost + \\ &+ \sum_{Q_j} (sz_j \times len_j \times cost_base_sm + 2 \times cost_in) \end{aligned} \quad (5.2)$$

We choose the lowest cost solution out of all single processor mappings and multiple processor mappings. This can be done as given in Equation 5.3. If a single processor mapping $SPM(m, n, PR_k)$ is chosen, then a new instance of PR_k is created, otherwise solution around selected pivot is accepted as $PM[m, n]$. Once matrix PM is fully computed, $PM[0, |T| - 1]$ (where $|T|$ is the number of processes) gives us the solution.

$$\begin{aligned} PM[m, n] &= \min_cost \{ \min_cost_{0 \leq k < |PR|} SPM(m, n, PR_k), \\ &\min_cost_{m < p < n} \{ MPM(m, p, n) \} \} \end{aligned} \quad (5.3)$$

Table 5.1 shows partial map (PM) matrix for *adjacent_process_list* of Figure 5.3 assuming that there are three types of processors PR_1 , PR_2 and PR_3 . We note that partial map matrix is basically an upper triangular matrix. Each valid entry is a 2-tuple. First field of this tuple is the pivot as explained above and second field is the processor type (-1 means that corresponding partial map requires multiple processors). We start from $PM[0, |T| - 1]$ and proceed until we find valid processor type for each partial map in the path. In this example, it reduces to *Sub-tree 2* of Figure 5.4.

5.1.4 Merging of shared memories

Partial map computed in Equation 5.3, produces a solution in which a separate instance of shared memory module is created for each queue not mapped to any local memory. This results in a more expensive interconnection network. Hence, in step 4 of Algorithm 1, we

	0	1	2	3	4	5
0	0, PR_1	1 , PR_2	1, -1	1 , -1	2, -1	3 , -1
1	NA	1, PR_1	1, -1	2, -1	2, -1	3, -1
2	NA	NA	2, PR_1	3 , PR_3	2, -1	3, -1
3	NA	NA	NA	3, PR_1	4, -1	4, -1
4	NA	NA	NA	NA	4, PR_1	5 , PR_1
5	NA	NA	NA	NA	NA	5, PR_1

Table 5.1: Partial map matrix

create new mappings by pairwise merging of shared memory instances in the given mapping under performance constraints. For example, consider a merger of both shared memories shown in Figure 5.2. This merger can be done by putting all the queues mapped on these shared memories into one of them and deleting the other such that performance constraints are not violated. This will allow us to remove two links and simplify the interconnection network.

5.1.5 Algorithm Complexity

Total number of entries in partial map matrix PM are $\frac{|T|^2}{2}$. The single processor mapping (SPM in Equation 5.3) is evaluated for all the processors available in the component library ($|PR|$). Further, as per Equations 5.1 and 4.11, this evaluation will take $O(|T| + |Q|)$ steps. Here $|Q|$ is the total number of queues in the process network. So, complexity of the first term in Equation 5.3 is $O(|PR| \times (|T| + |Q|))$. Now, the number of pivots in Equation 5.3 are at most $|T|$. Further, evaluation of a solution at any pivot can be done in $O(|Q|)$ because in equation 5.2, at most $|Q|$ queues need to be checked. Hence, complexity of the second term in Equation 5.3, is $O(|T| \times |Q|)$. Therefore, the overall time complexity of computing partial map is $O(|T|^2 \times (|PR| \times (|T| + |Q|) + |T| \times |Q|))$. Since value of $|PR|$ is small, complexity of computing partial map is bounded by $O(|T|^3 \times |Q|)$.

Now, number of pairs at any stage of shared memory merging is no more than $|Q|^2$ and there cannot be more than $|Q|$ merges. Further, checking performance constraints as per Equation 4.11 is bounded by $O(|T| + |Q|)$. So time complexity of shared memory merging stage is $O((|T| + |Q|) \times |Q|^3)$. Hence, Algorithm 1 has complexity of $O(|T|^3 \times |Q| + |T| \times |Q|^3 + |Q|^4)$ which becomes $O(|Q|^4)$ as typically $|T| \leq |Q|$.

5.2 Iterative Refinements of Initial Solution

In the previous stage, we considered processes only in the order in which they are present in the *adjacent_process_list*. This makes the quality of initial solution sensitive to this list. Further, changing mapping of a process might not only lead to using a lower cost processor, but also reducing the cost of interconnection network as the mapping of queues will also get affected. Hence, iterative refinements are required in order to optimize the architecture further.

In this Section, we describe the iterative refinement of initial solution. We will first describe the kind of refinements permitted.

Merging of shared memories

Consider a merger of both shared memories shown in Figure 5.2. This merger can be done by putting all the queues mapped on these shared memories into one of them and deleting the other such that performance constraints are not violated. This will allow us to remove two links and simplify the interconnection network. In this refinement step, given a mapping, we create new mappings by pairwise merging of shared memories instances in the given mapping.

Process migration and mutual exchange of process pairs

In these refinements, either a process is migrated from its current location to some other processor or processes change their processors in a pairwise manner. The benefit could come in the form of deletion of a processor and/or some links because it would change the mapping of queues onto any shared memory. Further, if due to exchange, there is a need to map some of the local memory mapped queues to shared memory, then we create a new shared memory instance for each one of them. We can later apply shared memory merging to make interconnections simpler.

Iterative refinement shown in Algorithm 2 starts with assigning initial solution as root of the search tree such as one shown in Figure 5.5. We prepare sorted list of feasible *candidates* in increasing order of cost from shared memory merging, process migration and mutual exchange of process pairs refinements. Then we control the search process with 2 parameters: maximum number of candidates to be selected (*max_breadth*) at current node in the search tree and maximum depth (*max_depth*) to look in into the current sub-tree in depth search manner without any improvement in the solution.

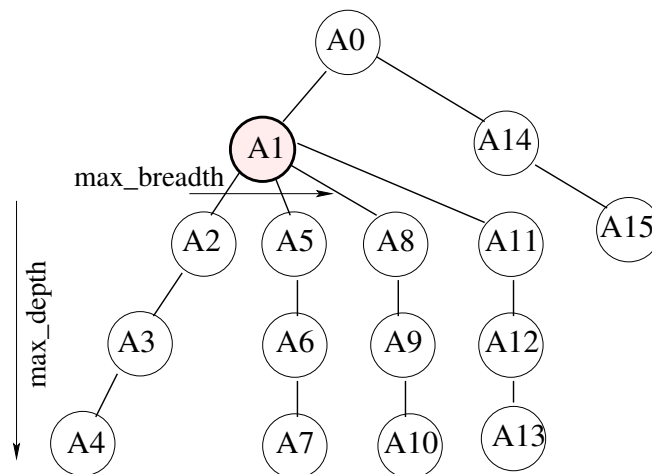


Figure 5.5: Iterative Refinement Tree

Considering example in Figure 5.5, A1 is the node under consideration. We prepare

Algorithm 2 iteratively_refine

```

1:  $root = best\_mapping = search\_path.leaf = PM[0, n]$ 
2: repeat
3:    $current\_node = search\_path.leaf$ 
4:   while ( $(search\_depth < max\_depth)$  and solution doesn't improve) do
5:     Increment  $search\_count$ 
6:     Prepare sorted list of feasible candidates in increasing order of cost from shared
       memory merging, process migration and mutual exchange of process pairs refine-
       ments
7:     Select first  $max\_breadth$  elements from candidates as  $current\_node.children$ 
8:     if  $current\_node.children \neq \Phi$  then
9:       Increment  $search\_depth$ 
10:      Insert first child of  $current\_node$  in the sub-tree path
11:      Update  $best\_mapping$  and accept sub-tree path in  $search\_path$  if solution is
        improved
12:     else
13:       Remove  $current\_node$  from search path
14:     end if
15:   end while
16:   Remove sub-tree path if solution is not improved in  $max\_depth$  search
17: until  $search\_count > max\_iterations \vee root.children == \Phi$ 

```

new mappings from A1 and select $max_breadth$ mappings from the sorted *candidates* as A1's children. If we see any improvement in the first sub-tree, then the whole sub-tree is appended to the search tree, otherwise we delete the whole sub-tree after refining max_depth . For example, if nodes A2, A3 and A4 don't improve the current best solution, the whole sub-tree $A2 \rightarrow A3 \rightarrow A4$ is deleted from the search path and new sub-tree starting from node A5 is considered. This procedure is repeated either till $max_iterations$

searches have been done or there is no new candidate.

5.3 Power Optimization

Some processors have the dynamic voltage scaling (DVS) feature [47]. A processor having DVS feature, can operate at different voltage levels and can switch from one voltage level to another dynamically. Typically, each voltage level corresponds to different clock frequency and higher voltage level results in higher clock frequency (lower clock period). As shown in Figure 5.6, higher operating voltage results in higher performance, but this also leads to more energy consumption.

In our MpSoC synthesis framework, we exploit the DVS feature of the processor for power optimization. In Section 5.1.3, we discussed how partial maps are built to construct the solution. In this technique, while evaluating whether a set of processes can be mapped to one processor, we try to assign different clock periods to various processes such that performance constraints are satisfied and energy consumption is reduced. Essentially, here, we perform static allocation of clocks for each process in the application. Further one might get more power savings if dynamic allocation of clocks also happen at runtime. However, this problem is out of scope of this thesis.

Towards power optimization, Algorithm 3 can be called at the **”Solution 1: Single processor mapping”** stage of the partial map computation. As we discussed above, when processor operates at higher voltage (lower clock period), energy consumption increases. So, the basic idea behind Algorithm 3 is to select a process and assign it the clock period such that it results in the least increase in the energy consumption. We first initialize clock period for each process to the clock instance of the processor corresponding to minimum voltage level. Referring to Figure 5.6, we note that this clock period is nothing but the maximum clock period (minimum frequency) of the processor. After initialization, we refine assignments of the clock period to processes until either performance constraint is

satisfied or maximum number of iterations are reached. Here, we have used Equation 4.10 to evaluate the performance constraints. In Algorithm 3, $nr_clk_instances$ is the number of clock period instances of processor PR_k . We note that a process can be assigned only one clock period out of $nr_clk_instances$. Hence, the maximum number of assignments which we investigate is the number of processes multiplied by $nr_clk_instances$ ($iter_limit$ in Algorithm 3).

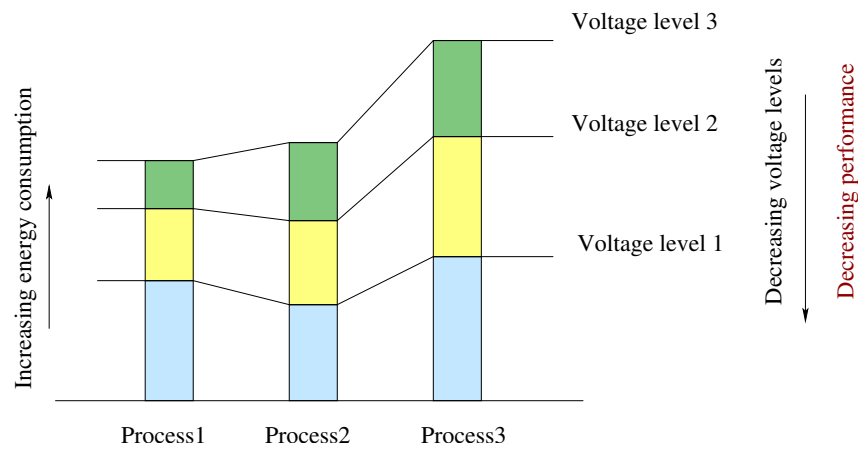


Figure 5.6: Voltage scaling in a processor

Table 5.3 illustrates Algorithm 3 using the inputs of Table 5.2. Entries of the first column of the Table 5.2 are processes. Other columns correspond to the energies consumed at different clock periods. An entry in these columns represent the energy consumed by a process during its execution as per throughput constraints for the process. For example, if the process T_1 needs to go through N_1 iterations to meet performance constraint and en_{10} is the energy consumed by T_1 for its single iteration at clock period C_0 , then $N_1 \times en_{10}$ is 100 energy units (some subunit of Joule). In Table 5.2, clock periods $C_0 > C_1 > C_2$ correspond to the increasing voltage levels. As discussed above, we first assign the maximum clock period to each process. This corresponds to the *Initialization* of Table 5.3. In this case performance constraints are not satisfied. So, we pick the process T_3 which leads to the least increase in total energy consumption if a smaller clock period is assigned to this

Algorithm 3 $\text{power_optimize}(\text{adjacent_process_list}(m, n), PR_k)$

```

1:  $nr\_clk\_instances =$  number of clock period instances of  $PR_k$ 
2:  $iter\_limit = (m - n + 1) \times nr\_clk\_instances$ 
3:  $cur\_iter = 1$ 
4: Initialize clock period for each process to the clock instance corresponding to minimum
   voltage level
5: while  $cur\_iter \leq iter\_limit$  do
6:   if performance constraint is satisfied then
7:     Return clock period instance assigned to each process
8:   else
9:     Decrement clock period for the process which leads to the minimum increase in
       the energy consumed by the process list  $\text{adjacent\_process\_list}(m, n)$  on processor
        $PR_k$ 
10:   end if
11: end while
12: if performance constraint is not satisfied then
13:   Mapping of  $\text{adjacent\_process\_list}(m, n)$  onto  $PR_k$  is infeasible
14: else
15:   Return clock period instance assigned to each process
16: end if

```

process. However, performance constraints are still not satisfied. So this step is repeated and the process T_1 is chosen next. Now, at iteration 2, performance constraints are satisfied, so clock period assignment at this iteration is returned.

Process	energy at clock period C_0	energy at clock period C_1	energy at clock period C_2
T_1	100	130	150
T_2	120	160	190
T_3	80	100	130

Table 5.2: Energy consumed by processes on processor PR_k at different clock periods

	T_1	T_2	T_3	Total energy	Performance satisfied
Initialization	$C_0, 100$	$C_0, 120$	$C_0, 80$	300	NO
iteration 1	$C_0, 100$	$C_0, 120$	$C_1, 100$	320	NO
iteration 2	$C_1, 130$	$C_0, 120$	$C_0, 100$	350	YES

Table 5.3: Example illustrating steps of Algorithm 3

5.4 Correctness of Synthesized Architecture

During the whole synthesis, the process network undergoes a set of refinements. Queues of the process network get mapped to hardware (memories) and processes are mapped to processors by compiling code for that specific processor. Essentially, multiple processes might be sharing the same processor and multiple queues might be sharing the same memory. So, arbitration is required at these architectural resources for implementing the correct functionality of the original application. This arbitration is provided by a dynamic scheduler. As long as this scheduler performs fair scheduling and follows the KPN communication semantics, the synthesis would be functionally correct. Synthesis of dynamic scheduler is not part of this thesis and may be taken up as the future work as indicated in Chapter 8.

5.5 Summary

In this Chapter, we described a heuristic based framework for MpSoC synthesis. This framework essentially consists of performing resource allocation and application mapping in two stages. We first generate the solution using a dynamic programming algorithm for cost as well as power optimization and then perform iterative refinements. In Chapter 7, we discuss some of the experimental results using this framework.

Chapter 6

Random Process Network Generator (RPNG)

In this Chapter, we present a user controllable pseudo random process network generator (RPNG). It generates random process networks which can be used as test cases for tools related to application specific multiprocessor architectures. RPNG generates database of computation and communication attributes of process networks for various processors and communication resources. These attributes are controlled by user specified parameters. RPNG also generates code for the process network ensuring that these networks are deadlock free. Generated code can be used as a workload either on a simulator or on an actual platform to study the underlying architecture. Another advantage of RPNG is that it enables one to reproduce results of other researchers.

6.1 Introduction

Many streaming media applications such as MPEG2, MPEG4, H264 etc. can be represented as Kahn Process Networks (KPN) [38, 21]. In KPN, nodes represent processes and arcs represent first in first out (FIFO) communication between processes. This represen-

tation of the application allows one to expose functional and data parallelism available in the application quite well. Figure 6.1 shows KPN model of MPEG-2 video decoding [55] application as an example.

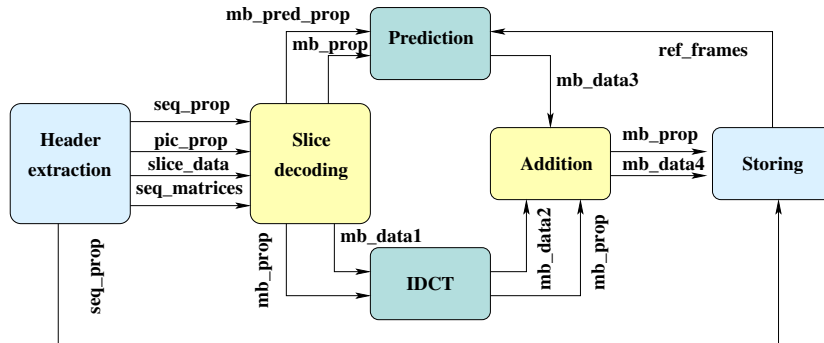


Figure 6.1: MPEG-2 video decoder

Effectiveness of KPN representation for streaming media application is well established in literature [21, 63, 31, 20, 70]. Unlike TGFF [22] and STG [71] which generate pseudo random directed acyclic task graphs for scheduling problems, there is no effort (to the best of our knowledge) towards having a set of benchmarks in KPN form which enables research in application specific multiprocessor synthesis for process networks. The objective of RPNG is to fill this gap.

Rest of the Chapter is organized as follows. In Section 6.2, we discuss why it makes sense to have a tool such as RPNG for generating process networks randomly and highlight our main contributions. Typical structure of a process network is discussed in Section 6.3. Procedure for generating topology of the process network is discussed in Section 6.4. Section 6.5 describes in detail how we insert computation and communication within a process and generate code and Section 6.6 describes generation of the database for application specific multiprocessor architecture synthesis problem instance. In Section 6.7, we summarize RPNG.

6.2 Motivation for RPNG

To synthesize the architecture for applications starting from the process network representation, a designer can take one of two possible approaches. The first approach relies on extracting periodic directed acyclic graph (DAG) out of the application and performing synthesis by static scheduling, resource allocation and binding. We refer to this approach as *static scheduling based approach*. This approach has extensively been studied in the literature from cost [19, 14] as well as power optimization [47, 66] point of view. DAG of the application can be extracted by decomposing and unrolling inner loops as discussed in [14, 40]. For example, Figure 6.2 shows part of the DAG obtained after decomposing the process network of MPEG2 decoder shown in Figure 6.1 at the granularity of a picture. As pointed out in [14], we also note that sub-processes (such as T1_1 in Figure 6.2) derived from the same process, should be mapped to the same resource due to strong internal communication (variable sharing etc.). This constraint has been termed as *type consistency constraint*.

The second approach tries to synthesize the architecture by exploiting the fact that *type consistency constraint* anyway needs to be respected, Hence, it uses higher level characteristics of the process network without decomposing it. In this, resource allocation and binding is performed and presence of a dynamic scheduler is assumed to take care of runtime scheduling. We refer to this method as *partitioning based approach* [63, 20, 25, 24].

Let us consider MPEG-2 video decoder shown in Figure 6.1 to compare two synthesis approaches discussed above. Processes *IDCT*, *Prediction*, *Addition* and *Storing* operate at macroblock level. Figure 6.2 shows part of the decomposed process network in the form of DAG and Figure 6.3 shows the corresponding pseudo-code. If this decoder performs decoding of frames of size 512×512 , then there will be around $(512 \times 512)/(64 \times 6) = 682$ macroblocks assuming 4:2:0 chroma format. Now we observe that if we try to derive DAG by unrolling inner loops of Figure 6.3, there will be more than $682 \times 4 = 2728$ computation

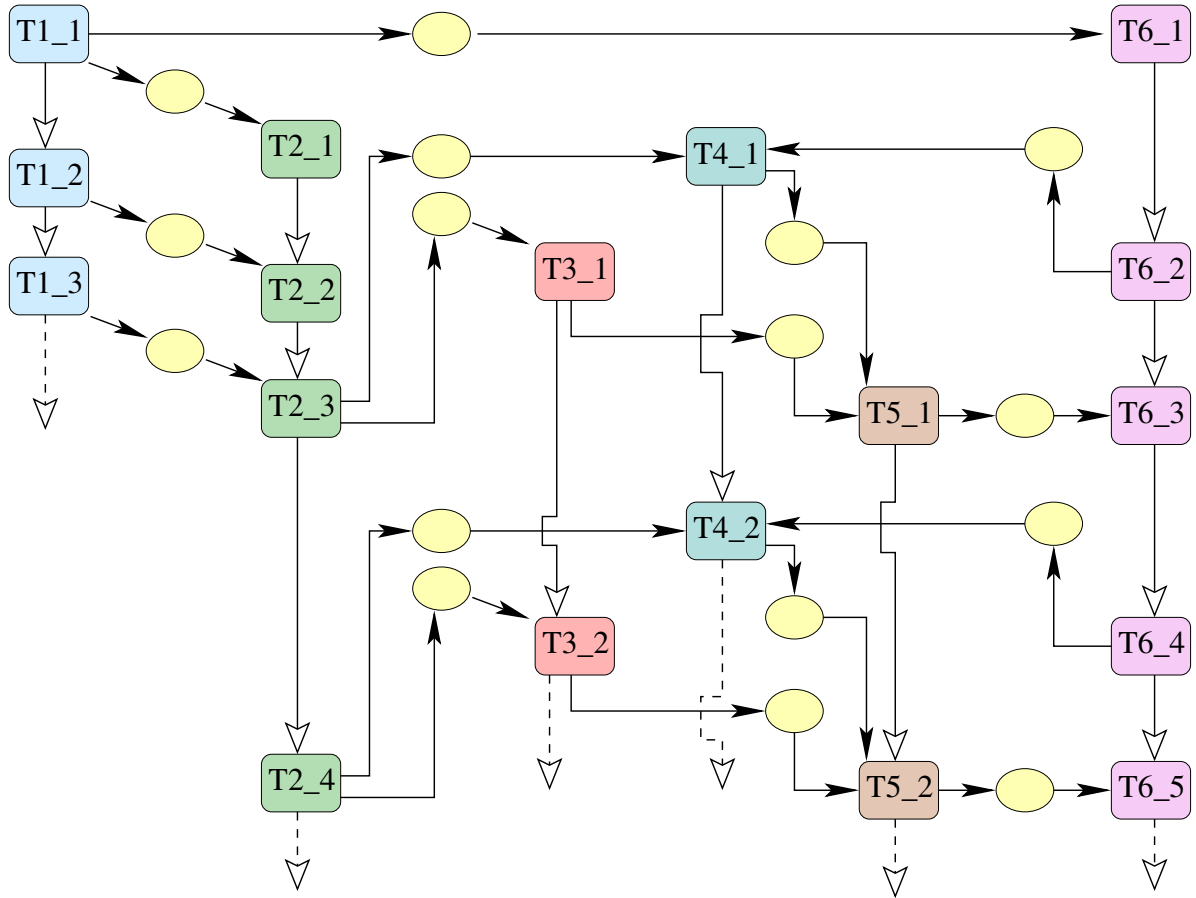


Figure 6.2: Unrolled MPEG-2 video decoder

sub-processes and $682 \times 6 = 4092$ communication sub-processes. Hence, total sub-processes (n) will be more than 6820.

If we choose algorithm proposed in [14] as a representative of *static scheduling based approach*, then this algorithm can solve the above problem in $n.Q.K^m$ time. Here n is the total number of sub-processes, Q is the bound on critical path delay, K is the maximum number of mappings for any sub-process and m is a number such that $0 \leq m \leq r \leq n$. Here r is the number of processes in the original application (6 in this example). It can be easily seen from Figure 6.2 that there will be at least 682 sub-processes on the critical path. Hence critical path delay Q will be $C_1 \times 682$ where C_1 is some constant ≥ 1 . Assuming

```

do_mpeg2_frame_decoding {
    header extraction;
    for each macroblock {
        do variable length decoding; do IDCT; do prediction;
        add IDCT and prediction results;
        store macroblock in the frame buffer;
    }
}

```

Figure 6.3: Computation within MPEG-2 video decoder

$m = r/2$ and $K = 4$, time taken T_1 by Chang's algorithm proposed in [14] will be:

$$\begin{aligned}
 T_1 &= 6820 \times C_1 \times 682 \times 4^3 = C_1 \times 640 \times 682^2 \\
 &\Rightarrow T_1 > C_1 \times (640)^3
 \end{aligned}$$

On the other hand, if we follow *partitioning based approach* by using an algorithm such as one proposed in [24], it takes only $T_2 = (N_Q)^4$ running time. Here N_Q is the number of channels in the original process network (15 in the above example). Clearly T_1 is much larger than T_2 . Now it can be noticed that *partitioning based approach* to synthesize the architecture for process networks is quite suitable due to the following reasons.

- Partitioning based synthesis algorithms run faster because of the much smaller problem size.
- It enables the designer to explore larger design space.

To enable evaluation of *partitioning based approach* for architecture synthesis, we need a tool which can generate a set of benchmarks in process network form as input to these approaches and serves the following objectives.

- Creating many problem instances for synthesis of architectures for process networks enabling research in this domain

- Producing benchmarks as workload for multiprocessor simulator or a real platform to study the underlying architecture
- Enabling researchers to reproduce results of others

RPNG accepts a set of parameters such as upper bounds on number of processes, on number of input and output channels of a particular process, amount of nesting of loops in a process etc. and generates the process network randomly alongwith the database. Generated database is nothing but a test case for architecture synthesis based on process network. This database essentially consists of computation and communication characteristics of the process network on processors and communication resources. RPNG also generates source code of the process network which is compatible with our thread library. The code generation phase of RPNG makes sure that process network is deadlock free. Using RPNG, a large number of random process networks can be generated quickly which allows one to effectively evaluate architectural synthesis algorithms and reproduce results of other researchers.

6.3 Structure of a Process Network

All the processes of the process network can be categorized into three classes: primary input (PI), intermediate and primary output (PO) processes. PI processes are the one which read data tokens from external world and initiate data flow inside the process network. Intermediate processes communicate with other processes only through internal channels, whereas PO processes act as sink and write data tokens to the external world. PI, intermediate and PO processes together define the order in which data flows in the process network. A sample process network is shown in Figure 6.4 where various PI and PO processes can be identified. In Section 6.5, we describe in detail how this classification of processes helps to generate deadlock free code for the process network.

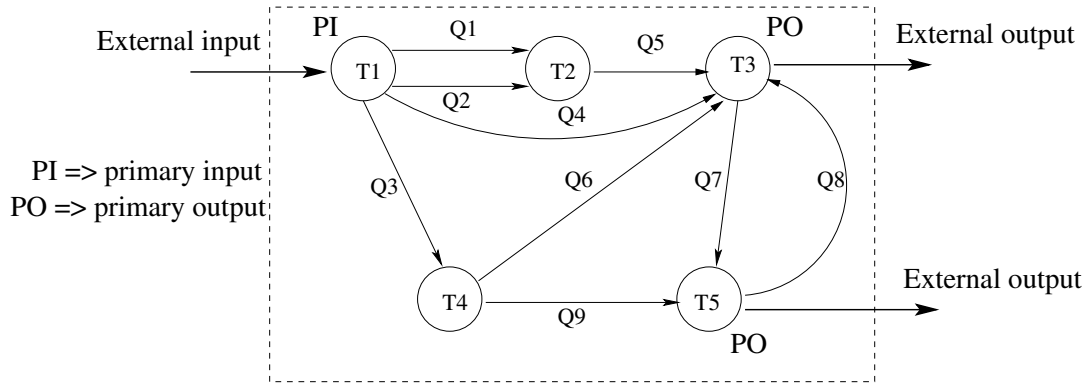


Figure 6.4: Example process network

As discussed above, PI processes are responsible for initiating data processing and communication. Intermediate processes further perform additional processing and data eventually reaches PO processes. It means that data flow inside the process network is from PI processes to PO processes which essentially implies that there is always some path from PI to PO processes. This property, which we term as *path property* must be ensured while adding new processes and channels during process network generation.

A process in the process network of a streaming application is nothing but a nested loop in which outer one is an infinite loop due to streaming nature of the process. Inside each loop statement (*LoopStatement*), computation and communication are interleaved. Figure 6.5 shows a sample process. *ComputeStatement* is some processing on internal data of the process. These are the statements which introduce finite computation delays. *CommunicateStatement* is a communication on some FIFO channel of the process network. This communication is either reading a token from an input channel (READ statement) or writing a token into an output channel (WRITE statement). *ComputeStatement*, *CommunicateStatement* and *LoopStatement* can appear in any order in a generic process.

Channels of the process network are FIFO in nature. In KPN, writes into the channels are non-blocking which could potentially lead to unbounded memory requirements. However, in practice, size of channel is limited. Hence, while generating the process network, we also specify randomly, maximum number of tokens which can reside in a channel. This

```
LoopStatement {  
    ComputeStatement;    CommmunicateStatement;  
    LoopStatement {  
        ComputeStatement;    CommmunicateStatement;  
        ... }  
    ... }  
}
```

Figure 6.5: Structure of a process

makes RPNG generated process network WRITE blocking as well. The code generation phase of RPNG takes this into consideration as well and makes sure that it does not introduce deadlock.

The problem of deadlock detection in a KPN is not decidable in general. However, the class of KPN, which confirm to the structure as generated by RPNG will not have deadlock. Deadlocking is indeed a dynamic behavior and synthesis methodology presented in the thesis exploits only the static structure of the process network.

6.4 Process Network Generation

RPNG uses Algorithm 4 to generate a standalone process network. The basic ideas behind this algorithm are as follows.

- By construction, ensure *path property* such that each process contributes towards dataflow from some PI to some PO process.
- Allow a channel to be present between any two processes. This leads to generation of a cyclic multi-graph structure of the process network as discussed earlier.
- Insert statements during code generation in a manner such that at any point in time, at least one process is ready to read/write data to/from a channel. This makes sure that the generated process network is deadlock free.

Based on the above ideas, RPNG works in three phases: creating process network topology, inserting computation and communication statements in the processes which helps in generating code and creating database for the synthesis problem instance. In Algorithm 4, steps 1-5, 6 and 7 correspond to these three phases respectively. The first phase is discussed in this Section and rest of the phases are described in Sections 6.5 and 6.6 respectively.

Algorithm 4 createPN

- 1: *initPN*: initialize the process network.
 - 2: **repeat**
 - 3: *addProcess*: insert a new process alongwith associated channels.
 - 4: *addChannel*: select *doFanout* (connecting output channels of some process) or *doFanin* (connecting input channels of some process) with equal probability.
 - 5: **until** upper bound on number of processes present in the process network (*ub_num_processes*) is reached
 - 6: *generateCode*: insert READ and WRITE communication statements in processes and dump source code of the process network.
 - 7: *createDatabase*: generate computation and communication characteristics of the process network for architectural resources based on the user specified attributes.
-

Various phases of process network generation are controlled by a set of parameters. Parameters which control the topology of the process network and code generation are as follows. Other parameters and attributes are discussed in detail in Section 6.6.

- **num_pi**: Number of PI processes which act as sources in the process network.
- **ub_num_po**: Upper bound on number of PO processes. These are the sink processes. In the final process network, there will be at least one PO process.
- **num_processes**: Number of processes present in the process network. This must be $\geq (\text{num_pi} + \text{ub_num_po})$.

- **ub_num_in_chns:** Upper bound on the number of input channels of a process.
- **ub_num_out_chns:** Upper bound on the number of output channels of a process.
- **ub_nesting_level:** Upper bound on the number of nested loops within any process.

6.4.1 Initialization of Process Network

In step 1 (*initPN*) of Algorithm 4, we instantiate `num_pi` PI processes and `ub_num_po` PO processes. At this stage, it is required that there is at least one path from a PI to some PO process. To create this path, we choose an unconnected PI (say P_i) and another process from the set of connected PI and all PO processes (say P_j is chosen) randomly with uniform probability. At this point a new channel is instantiated with reader as P_j and writer as P_i . Thus we establish paths from each PI to some PO process.

<code>ub_num_processes</code>	<code>num_pi,</code> <code>ub_num_po</code>	<code>ub_num_in_chns,</code> <code>ub_num_out_chns</code>
6	2, 2	3, 3

Table 6.1: Parameters for process network graph generation

Figure 6.6(a) shows the structure of a process network after *initPN* for input parameters specified in Table 6.1. In Figure 6.6, PI_i is i^{th} PI process, PO_j is j^{th} PO process, $IntP_k$ is k^{th} intermediate process and C_l is l^{th} channel considered so far. Since initial path from a PI process to some PO process could be either direct or through some other PI process, some of PO processes might remain unconnected. PO_1 is one such PO process in Figure 6.6(b). If there is any such process, we remove it from the process list. It can be easily observed that there will be at least one PO process which is connected to some PI process after *initPN* step.

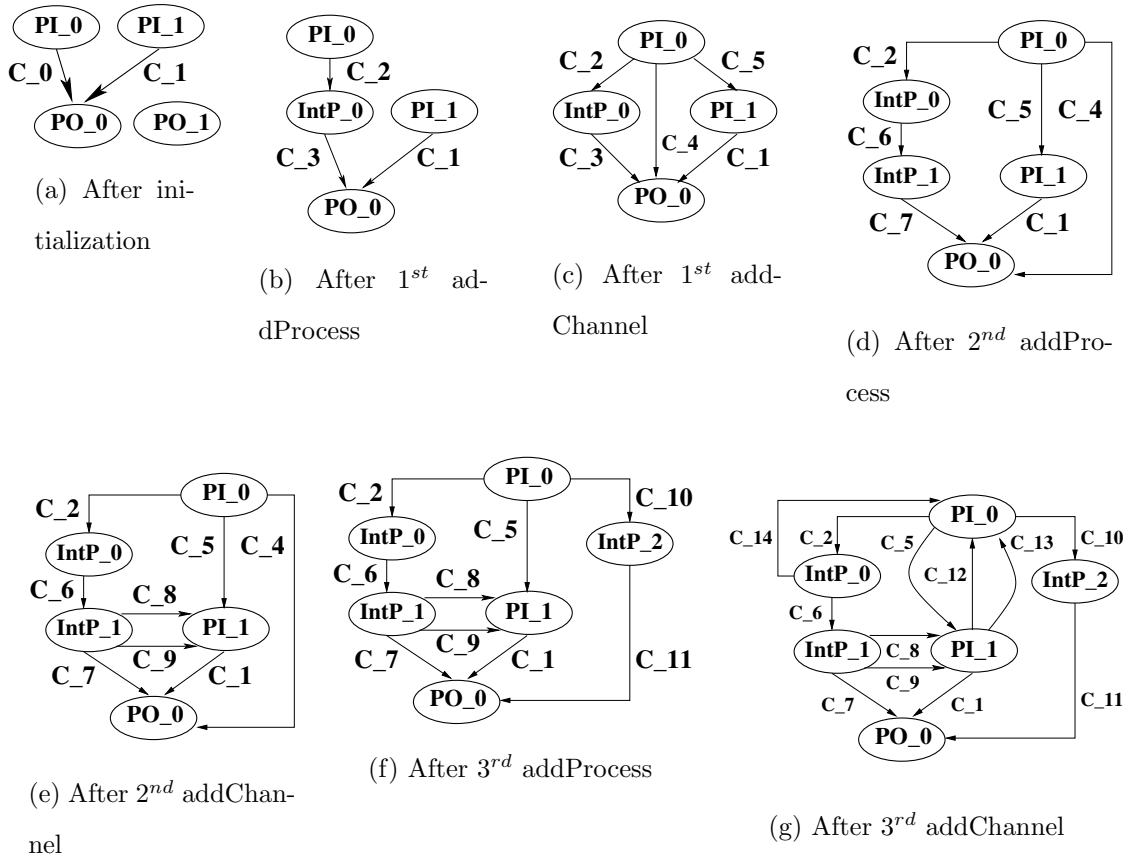


Figure 6.6: Various stages of process network graph generation

6.4.2 Addition of a New Process

In step *addProcess* of Algorithm 4, to add a new process, one of the channels is chosen out of all the channels with uniform probability. Now if a new node (P3) is inserted between two already existing nodes say P1 and P2, then the channel P1→P2 is removed and two additional channels are added P1→P3 and P3→P2. Since P1 is reachable from some PI (due to initialization), P3 will also be reachable because of channel P1→P3. Similarly, since there exists a path from P2 to PO, a path will also exist from P3 to PO because of the channel P3→P2. Thus while adding new processes, *path property* remains valid.

Figure 6.6(d) shows an instance of *addProcess*. Channel *C_3* is selected with uniform probability out of all the channels present in the process network and broken into two

parts. C_3 is removed and two new channels (C_6 and C_7) alongwith a new intermediate process $IntP_1$ are introduced.

6.4.3 Addition of a New Channel

Though a channel is removed and two new channels alongwith a new process are added in *addProcess* step, more channels are added mainly in step 4 (*addChannel*) of Algorithm 4. *doFanout* and *doFanin* have certain similarity with *fanout* and *fanin* steps of TGFF [22] respectively. Unlike TGFF where *fanout* and *fanin* steps result in addition of new nodes also, we don't add new processes in *addChannel*, but only establish connections between unutilized input and output ports of processes. The reason for not adding new processes in this step lies in the structure of process network which could be a cyclic multi-graph. Our approach leads to introduction of back edges, whereas TGFF approach always generates directed acyclic graph (DAG).

Algorithm 5 describes *doFanout* step of *addChannel*. Figures 6.6(c) and 6.6(e) show addition of new channels after *doFanout*. Table 6.2 shows fanout probabilities corresponding to step 6 of Algorithm 5. To select a process such that its unused output channels can be connected, a random number with uniform probability is generated in the range $[0, 1]$. In this case, this number was 1 and hence as per Table 6.2, process $IntP_1$ was selected and channels C_8 and C_9 were added.

Similar to *doFanout*, Algorithm 6 describes *doFanin* step of *addChannel*. Table 6.3 shows fanin probabilities as computed in steps 2-5 of Algorithm 6. Now we need to select a process which will get its unused input channels connected (step 6 of Algorithm 6). Again a random number was generated in the range $[0, 1]$ with uniform probability. At this particular step, 0.3333 was found. This led to selection of process PI_0 and channels C_{12} , C_{13} and C_{14} were added in the network of Figure 6.6(f). Resultant process network is shown in Figure 6.6(g).

Algorithm 5 doFanout

-
- 1: Compute total_available_output_links at all the processes
 - 2: **for all** processes **do**
 - 3: Compute available_output_links at this process
 - 4: Compute fanout probability of the process which is available_output_links/total_available_output_links
 - 5: **end for**
 - 6: Select source process (src_process) out of all the processes based on their fanout probabilities.
 - 7: Prepare the list of the processes other than src_process such that they have some unutilized input candidate channels.
 - 8: If there are not enough input links, then whatever is available should be connected. Otherwise, destination process should be chosen with a uniform probability distribution.
-

	PI.1	PI.0	PO.0	IntP.0	IntP.1
Unused out links	2	0	3	2	2
Fanout probability	0.222	0	0.333	0.222	0.222

Table 6.2: Fanout probabilities for Figure 6.6(d)

	PI.1	PI.0	PO.0	IntP.0	IntP.1	IntP.2
Unused in links	0	3	0	2	2	2
Fanout probability	0	0.333	0	0.222	0.222	0.222

Table 6.3: Fanin probabilities for Figure 6.6(f)

Algorithm 6 doFanin

- 1: Compute `total_available_input_links` at all the processes
 - 2: **for all** processes **do**
 - 3: Compute `available_input_links` at this process
 - 4: Compute `fanin` probability of the process which is `available_input_links/total_available_input_links`
 - 5: **end for**
 - 6: Select destination process (`dest_process`) out of all the processes based on their `fanin` probabilities.
 - 7: Prepare the list of the processes other than `dest_process` such that they have some unutilized output candidate channels.
 - 8: If there are not enough output links, then whatever is available should be connected. Otherwise, source process should be chosen with a uniform probability distribution.
-

6.5 Code Generation

6.5.1 Process Flow Level and Condition for Deadlock Free Execution

In Section 6.3, we discussed that the process network exhibits dataflow property i.e. all the processes contribute towards dataflow. Hence, in RPNG, we have introduced the notion of process flow level which primarily comes from the observation that PI processes are the ones which initiate data flow inside the process network and rest of the processes aid this data flow in some order. Process flow levels are assigned in *assignProcessFlowLevels* step of Algorithm 7. This step is a simple breadth first search with the PI processes being assigned the highest flow level and all the processes to which they write are one level lower than flow level of PI and so on. Processes once assigned are not considered again. Figure 6.7(a), shows how the process flow levels are assigned for the process network of Figure

6.7(b). Initially PI processes PI_0 and PI_1 are assigned flow level 2. Now reader processes of output channels of PI processes are IntP_0, IntP_2 and PO_0. So these processes are assigned flow level 1. Similarly reader process of output channels of processes at flow level 1 which have not been assigned flow level is only IntP_1. Hence this process is assigned flow level 0.

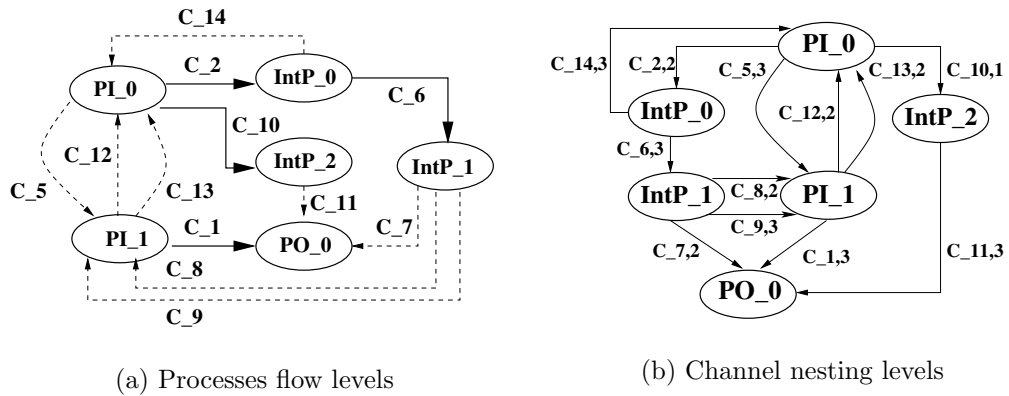


Figure 6.7: Nesting and flow levels in example process network

The order of READ and WRITE communication statements in each process is important as it would determine whether the process network generated will be deadlock free or not. A process network is deadlock free if at no point in time during its execution, all the processes block, either on READ or on WRITE communication statements. A process would block on a READ if the channel from which it is to read is empty and it would block on a WRITE if corresponding output channel is full. For a deadlock free execution, we have to ensure that at any time there exists at least one process which can read/write from/to some channel.

Since the process network exhibits dataflow, READ and WRITE communication statements must be inserted inside the processes by taking care of their flow levels. Let us classify these in order to evaluate valid sequences of communication statements.

1. **WR_SL** : Write to process which is at the same flow level or lower flow level.
2. **WR_HI** : Write to higher flow level process.

3. **RD_SL** : Read from a process which is at the lower flow level or same flow level.
4. **RD_HI** : Read from higher flow level process.

Example of one sequence of communication statements is {WR_SL, WR_HI, RD_LS, RD_HI}. We already discussed that apart from primary input (PI) processes, to aid dataflow inside the process network, any other process must first read data, perform some computation and then write data. This means that WR_SL and WR_HI cannot be the first communication statement in the sequence. RD_SL is also ruled out as the first communication statement, because if there is a cycle between reader and writer processes (such as cycle between PI_0 and IntP_0 in Figure 6.7(a)), both will keep on waiting for each other to write something in the channels before the read operation can be performed. So, RD_HI is the only possibility as the first statement of the communication sequence. Now we note that RD_SL cannot be the next communication because of the reasons discussed above. Hence, there are only two possible communication statement sequences: {RD_HI, WR_SL, WR_HI, RD_SL} and {RD_HI, WR_HI, WR_SL, RD_SL}. We term the constraint on the order of communication statements as *flow constraints*. We have used the first sequence in our implementation. Now, one can easily observe that for these communication sequences, each process will always have either something to read from or something to write into a channel. This makes sure that there will not be a deadlock. The *flow constraints* can be summarized as follows.

1. **RD_HI operation** : Since primary input (PI) processes are at the highest process level, they cannot perform RD_HI. So, the PI processes start communication by performing WR_SL operation. This makes sure that processes at lower levels are ready for RD_HI operation followed by WR_SL and WR_HI operations.
2. **WR_SL and WR_HI operations** : Since RD_HI is the first communication statement, a process will not be blocked forever on write operations.

3. **RD_SL operation** : RD_SL communication by an higher level process will be successful because the lower level process would have already gone through the RD_HI, "Compute" and WR_HI phases.

6.5.2 Insertion of Statements in Processes

As discussed in Section 6.3, a process is basically a nested loop with computation and communication taking place in these loops in an interleaved manner. Hence we associate a loop nesting level to each channel randomly in the range 1 to **ub_nesting_level** in step 6 (*assignNestingLevelToChannels*) of Algorithm 7. Nesting levels of channels of Figure 6.6(g) are shown in Figure 6.7(b) next to each channel's name when **ub_nesting_level** was 3.

Algorithm 7 generateCode

- 1: *assignNestingLevelToChannels*: randomly associate loop nesting level to each channel.
 - 2: *assignProcessFlowLevels*: assign data flow level to each process.
 - 3: **for all** processes **do**
 - 4: *insertLoops*: insert empty nested loops in the process depending on maximum nesting level of its channels.
 - 5: *insertChannelWrites*: insert WRITE communication statements in the process.
 - 6: *insertChannelReads*: insert READ communication statements in the process.
 - 7: **end for**
 - 8: *printCode*: dump source code of the process network.
-

Insertion of various statements in processes starts with inserting loops in step *insertLoops* of Algorithm 7. We find out maximum loop nesting of a process by looking at nesting levels of its input and output channels and insert that many loops. For example, maximum loop nesting in the process PL0 of Figure 6.7(b) is 3. We insert 3 nested loops for this process as shown in Figure 6.8. In RPNG, each *LoopStatement* is deterministic i.e. its lower and upper bounds are known. We also impose the restriction that bounds of a loop at any

nesting level must be same across all processes to make sure that there is neither underflow nor overflow of tokens inside any channel.

After inserting empty loops in processes, we insert WRITE communication statements. While inserting these statements, flow constraints 2 and 3 discussed above are taken care of. Figure 6.8 shows WRITE statements after *insertChannelWrites* step of Algorithm 7.

```

LoopStatement {
    WRITE into channel C_10;
    LoopStatement {
        WRITE into channel C_2;
        LoopStatement {
            WRITE into channel C_5;
        }
    }
}

```

Figure 6.8: WRITE statements in process PI_0

Insertion of READ statements in the processes takes place in *insertChannelReads* step of Algorithm 7. Here we need to make sure that insertions of READs are as per flow constraints 1 and 4 discussed earlier. So at any nesting level, READs from higher level processes are inserted before any WRITES and READs from lower level processes are appended after all the WRITES. Then we insert *ComputeStatements*. To give a feel of how final structure of processes look like, Figures 6.9 and 6.10 show statements of two processes at different flow levels communicating with each other for two processes of the process network shown in Figure 6.7(b).

We discussed above that in RPNG, each *LoopStatement* is deterministic and bounds of these statements are same at each nesting level across all processes. This is essential to make sure that reader processes consume the same number of tokens which writer processes have

```
LoopStatement {
    ComputeStatement;    WRITE into channel C_10;
    LoopStatement {
        ComputeStatement;    WRITE into channel C_2;
        LoopStatement {
            ComputeStatement;    WRITE into channel C_5;
            ComputeStatement;    READ from channel C_14;
        }
        ComputeStatement;    READ from channel C_12;
        ComputeStatement;    READ from channel C_13;
    }
}
```

Figure 6.9: Statements in process PI.0

produced. Though loops are deterministic, RPNG allows different loop bounds at different nesting levels.

Finally, step *printCode* of Algorithm 7 dumps code of the whole process network in C language compatible with our thread library. Our thread library is also written in C and internally it uses *pthread*s for parallelism. If generated code is used as a workload on a simulator or on an actual platform, then underlying threading mechanism *pthread*s should be replaced accordingly.

It can be noticed that a *ComputeStatement* is the one which imposes processing delays. In Section 6.6, we discuss input parameters in the form of a set of attributes which decide computation and communication structure of the process network. Out of these attributes, **stmt_processor_attribute** relate *ComputeStatement* statements of processes to processors for corresponding mapping. Examples of these attributes are *n_cyc_avg* and *n_cyc_var*, which are the average and variance for the number of cycles taken by some

```
LoopStatement {  
    LoopStatement {  
        ComputeStatement;    READ from channel C_2;  
        LoopStatement {  
            ComputeStatement;    WRITE into channel C_6;  
            ComputeStatement;    WRITE into channel C_14;  
        }  
    }  
}
```

Figure 6.10: Statements in process IntP_0

ComputeStatement on a processor respectively. By making use of these parameters, we assign average and variance for number of cycles taken by each *ComputeStatement* on a particular processor. Depending on the final architecture and application mapping, this information can be used by the run time schedulers to provide data dependent behavior for generated process network.

6.6 Creation of Database

RPNG allows a user to specify a number of attributes. These attributes are associated with the processes, channels and architectural resources and are used to generate computation and communication characteristics of the process network on various processors and communication resources. RPNG accepts the following types of attributes.

- **channel_attribute:** Attributes associated with each channel.
- **processor_attributes:** Attributes specific to a processor.
- **memory_attributes:** Attributes specific to a memory.

- **switch_attributes:** Attributes specific to a switch. In RPNG, a switch can be a cross-bar switch or a bus.
- **process_processor_attributes:** Attributes which relate processes to processors for corresponding mapping.
- **stmt_processor_attributes:** Attributes which relate *ComputeStatement* statements of processes to processors for corresponding mapping.

Similar to TGFF, we specify two parameters *av* and *mul* for each attribute apart from its name. We use the same equation given in TGFF to generate attribute value.

$$attrib = av + jitter(mul.rand, var) \quad (6.1)$$

where *rand* is a random value between -1 and 1, *var* is an input parameter in the range $[0, 1]$ and *jitter*(*x*, *y*) returns a random value in the range $[x.(1 - y), x.(1 + y)]$.

There are very few applications available in the process networks form. Due to this reason, exact probability distribution for each attribute could not be obtained. That is why we had to rely on an approach similar to TGFF to randomly generate values of different attributes. However, structure of RPNG is very general and one can easily refine RPNG to apply exact probability distributions for various attributes.

As an example, Figure 6.11 shows some of the attributes for the problem instance of application specific multiprocessor synthesis problem for the process networks. Here channel.attribute *thr* which is throughput (number of tokens/second on the channel), *channel_size* which is maximum number of tokens in a channel and *token_size* which is size of a token in any channel must be specified. All other channel attributes are optional. Here *thr* is nothing but the performance constraint on the application. Similarly stmt_processor_attributes *n_cyc_avg* and *n_cyc_var*, which are the average and variance for the number of cycles taken by some *ComputeStatement* on a processor, are required and

all other attributes are optional. Other attributes shown in Figure 6.11 are also necessary to be specified.

num_processors	3
num_memories	3
num_switches	3
channel_attribute	channel_size 4 2
channel_attribute	token_size 1000 100
channel_attribute	thr 500 100
processor_attributes	freq 500 50
processor_attributes	contxt 200 0
processor_attributes	cost_proc 100000 50000
processor_attributes	cost_base_lm 20 5
memory_attributes	bwmm 10 2
memory_attributes	cost_base_sm 15 4
stmt_processor_attributes	n_cyc_avg 200 50
stmt_processor_attributes	n_cyc_var 30 10

Figure 6.11: Sample attributes for database

Now for input parameters of Table 6.1 and attributes specified in Figure 6.11, partial database of 0th processor is shown in Figure 6.12. In this figure, *pr_cyc_avg* is the average number of cycles taken by a process for its single iteration on the corresponding processor. This was computed by accumulating delays of various statements present in the process. This partial database shows part of the problem instance of application specific multi-processor synthesis problem for the process networks. Thus we see that RPNG is capable of handling an arbitrary number of attributes and generating problem instances and test cases.

```
@Processor 0 {  
# freq  contxt  cost_proc  cost_base_lm  
448.35 200.00 55029.19 14.77  
#-----  
# process_id  n_cyc_avg  
0 2060.000000  
1 1883.000000  
2 1530.000000  
3 1745.000000  
4 1540.000000  
5 753.000000  
}
```

Figure 6.12: Partial database

6.7 Summary

We presented a tool for randomly generating process networks which can be used for generating test cases for tools related to synthesis of application specific multiprocessor architectures. Apart from generating computation and communication characteristics of the process network in the database, RPNG also produces the source code which executes without deadlock. This code can be used as a workload for simulation purposes to study the underlying architecture.

RPNG is highly parametrized and capable of handling a variety of computation and communication attributes. Using RPNG, a large number of process networks can be generated quickly which in turn allows one to comprehensively test the algorithms and also reproduce results of others. We have used RPNG extensively for performing experiments on our framework for synthesis of application specific multiprocessor architectures

for applications represented as process networks. RPNG has been implemented in C++ and will be publicly available from the site [27] soon.

Chapter 7

Experimental Results

In this Chapter, we describe the experimental results which demonstrate application of our MILP formulation, our MpSoC synthesis framework and RPNG. We essentially performed two sets of case studies, first on a real life application i.e. MPEG-2 decoder and then on a set of process networks generated using RPNG.

7.1 MPEG-2 Video Decoder Case Study

We implemented a multi-threaded library using which an application written in C can be modeled as KPN. We converted the sequential code of MPEG2 decoder into the KPN as shown in Figure 7.1. Process *Header extraction* finds out the header information of the video sequence. Process *Slice decoding* performs variable length decoding at slices. Similarly functionality of processes *IDCT*, *Prediction*, *Addition* and *Storing* are obvious from their names. These four processes work on macroblocks.

7.1.1 Using MILP Solver

In Chapter 4, we described two MILP formulations. One for resource allocation, application mapping onto the architecture and the other for getting the interconnection networks. In

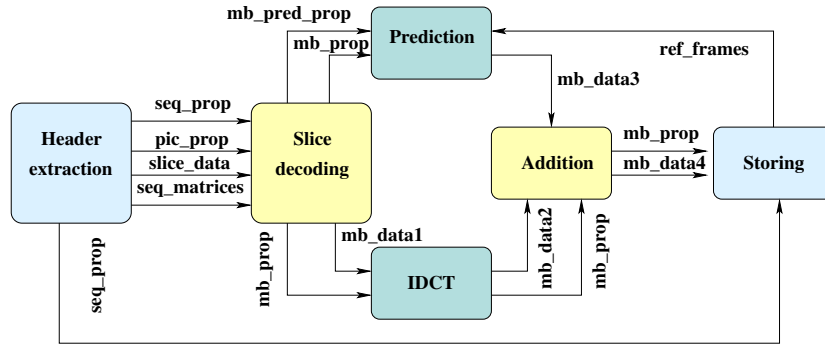


Figure 7.1: MPEG-2 Video Decoder Process Network

this Section, we discuss the experiment performed for MPEG2 decoder. In this experiment, we generated the MILP as presented in Chapter 4 and solved it using *LP Solve* [46].

Figure 7.2 shows the MPEG-2 decoder process network annotated with queue parameters. Here 5-tuple next to each edge is the parameter values for size of a token being transferred, the maximum number of tokens allowed in queue, throughput constraint, number of tokens produced or consumed by the writer or reader process in its single iteration respectively.

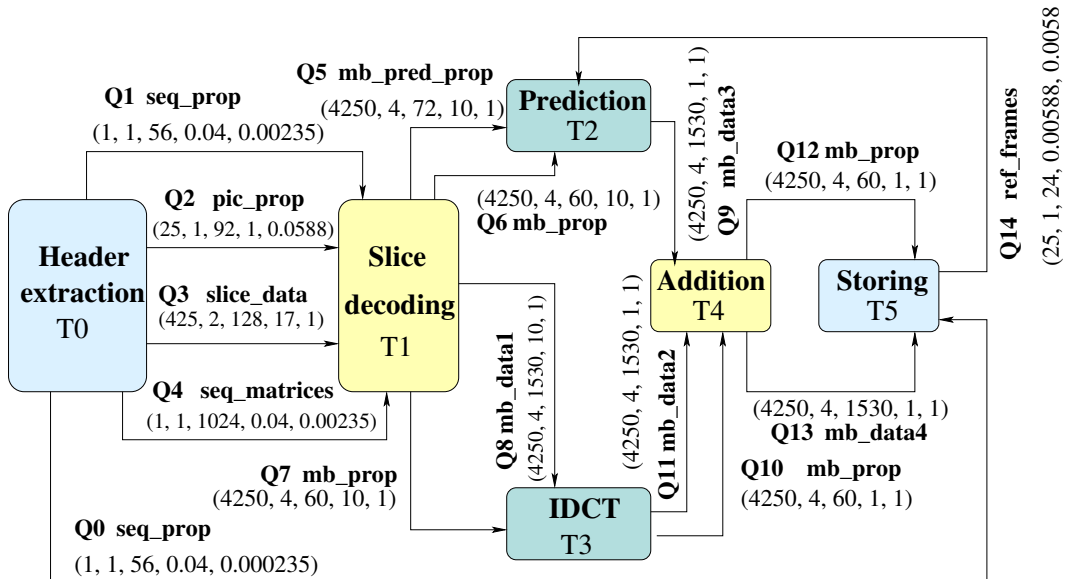


Figure 7.2: Annotated MPEG-2 video decoder process network

First part of Table 7.1 shows process parameters of this KPN. These parameters were obtained by following the procedure shown in Figure 2.5. The application was converted to the MachSuif [49] intermediate representation. Scheduling was performed for SPARC V8 [69] back-end written in the MachSuif. Schedule length for each basic block multiplied by the profile information of the same was used to obtain software execution estimates for the mentioned processor. An ideal cache was assumed. Though not very accurate, this flow provides first cut estimates.

Second part of Table 7.1 shows processor attributes. Each processor has associated cost (pr_cost_k). $ctxt_k$ is the context switch overhead on compute unit PR_k and $freq_k$ is its frequency. $cost_base_lm_k$ is the cost of mapping each unit size of the channel mapped onto local memory of PR_k .

The last part of Table 7.1 provides details of interconnection network (IN) parameters. $swty_m$ is the type of switch where, 1 indicates a shared bus and 0 a cross-bar switch. $cost_sw_m$ is cost of m^{th} switch SW_m . $cost_sw_in_m$ is the cost of the link from compute unit or memory to this switch. sp_m is the number of compute unit side ports and smm_m corresponding memory side ports. bws_w_m is its bandwidth. Relatively higher link cost and higher bandwidth can be correlated with the presence of multiple buses.

In this experiment, presence of three memory units was assumed with bandwidth of 800000 and base cost of 5. The synthesized architecture and process network mapping is shown in Figure 7.3 after solving the MILP using *LP Solve* [46]. It has three processors, two shared memory modules and switch SW_4 which is a cross-bar switch. As expected, channels mapped to the local memory have their readers and writers at the same processor. If the reader and writer of the channel go to different processors, then corresponding channel goes to the shared memory. In this architecture, IN is basically a cross-bar switch SW_4 due to higher throughput requirements. This example took less than 5 minutes on a workstation having Intel XEON CPU running at 2.20GHz and 1GB RAM.

Process parameters					
Number of cycles taken by a process T_i per iteration on processor PR_k for $\forall k$					
ncy_{0k}	ncy_{1k}	ncy_{2k}	ncy_{3k}	ncy_{4k}	ncy_{5k}
1043416	717331	75248	54564	23030	65134

Processor parameters				
PR_k	$contxt_k$	pr_cost_k	$freq_k$	$cost_base_lm_k$
PR_0	200	300000	100e+06	10
PR_1	200	700000	500e+06	10
PR_2	200	300000	100e+06	10
PR_3	200	500000	400e+06	10

Interconnection network parameters						
SW_m	$cost_sw_m$	$cost_sw_in_m$	sp_m	smm_m	bws_w_m	$swty_m$
SW_0	20000	2400	4	4	3e+06	1
SW_1	40000	4800	4	4	6e+06	1
SW_4	320000	2400	4	4	24e+06	0
SW_5	160000	2400	4	3	12e+06	0
SW_6	120000	2400	3	2	9e+06	0

Table 7.1: Inputs for MILP based solver

7.1.2 Using Heuristic Based Framework

In Chapter 5, we described the heuristic based MpSoC synthesis framework. In this Section, we discuss an experiment on MPEG-2 decoder using the synthesis framework demonstrat-

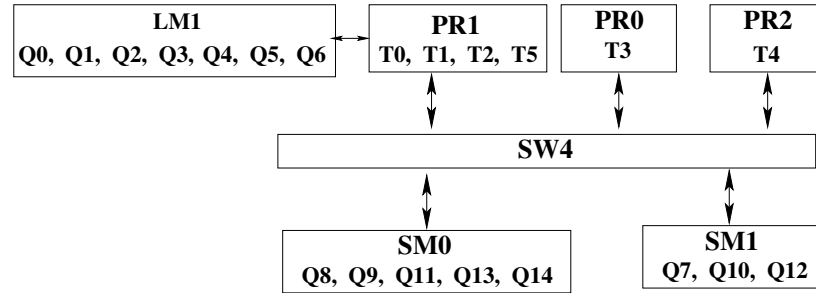


Figure 7.3: Synthesized architecture and mapping for Table 7.1

ing its usefulness to the system designer.

Figure 7.2 shows the process network annotated with queue parameters. First part of Table 7.2 gives other process parameters of MPEG-2 decoder KPN. These parameters were obtained by using the procedure described in [36] for ARM7TDMI [6] and LEON [30] processors. Second part of Table 7.2 show processor parameters. Each processor type (PR_k) has associated cost (pr_cost_k), context switch overhead on processor ($context_k$), its frequency ($freq_k$) and cost of mapping unit size of the queue onto local memory ($cost_base_lm_k$). Processors PR_0 and PR_1 are taken as different implementations of LEON. Similarly, PR_2 is an implementation of ARM7TDMI. Apart from this, presence of one type of memory unit was assumed with bandwidth of 5,000,000 bytes/second and base cost of 0.6 units. Link cost $cost_in$ was taken to be 5,000 units. Corresponding synthesis results are shown in Figure 7.4. We note that it is a heterogeneous architecture consisting of three instances of processor type PR_1 and single instance of PR_2 . Moreover, one shared memory was found to be sufficient. This also makes the interconnection network quite simple by instantiating only one shared bus.

7.2 Experiments Using RPNG

In Chapter 6, we discussed that one of the objectives of *Random Process Network Generator (RPNG)* was to create many problem instances for MpSoC synthesis problems enabling research in this domain. So, we generated a number of process networks using RPNG so

Process parameters						
Number of cycles taken by a process T_i per iteration on processor PR_k for $\forall k$						
k	ncy_{0k}	ncy_{1k}	ncy_{2k}	ncy_{3k}	ncy_{4k}	ncy_{5k}
0	666714	55969	32795	40087	32940	10036
1	666714	55969	32795	40087	32940	10036
2	978350	75550	44096	44087	46742	20449

Processor parameters				
PR_k	$contxt_k$	pr_cost_k	$freq_k$	$cost_base_lm_k$
PR_0	200	200000	150e+06	0.5
PR_1	200	400000	200e+06	0.5
PR_2	100	170000	133e+06	0.5

Table 7.2: Process and processor parameters

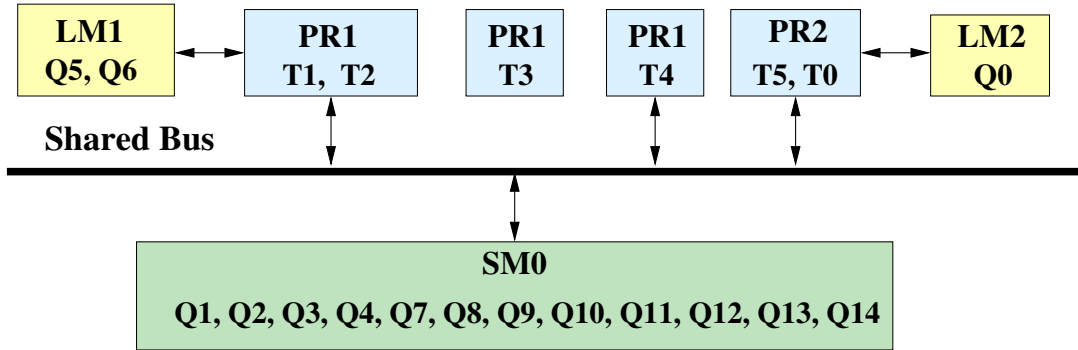


Figure 7.4: Synthesized architecture and mapping for Table 7.2

as to experiment the following.

- To test the scalability of MpSoC synthesis framework described in Chapter 5 i.e. whether this framework can produce solutions in reasonable time
- To test the quality of solutions produced by 1st stage (constructive phase) of MpSoC

synthesis framework against 2^{nd} stage (iterative refinements)

With above objectives, we performed a number of experiments using RPNG in which we applied process networks generated by RPNG as input to the MILP formulation and heuristic based MpSoC synthesis algorithm described in Chapter 5. RPNG can be used for these experiments in the following steps.

1. Provide process network and architecture attributes to RPNG for generating synthesis problem database
2. Apply the problem database obtained from the first step to the MpSoC synthesis framework

Let us go through an example to understand the above steps. Figure 7.5 shows various attributes which are given to RPNG for generating problem database. As discussed in Chapter 6, these attributes define the process network topology and application mapping characteristics. Though meaning of various attributes are evident from the name, details of the same can be seen in Section 6.6. Figure 7.6 shows the topology of the process network generated for the attributes of Figure 7.5. This process network is composed of 10 processes and 20 channels and also has couple of cycles. Figure 7.7 shows the part of the problem database for the same set of attributes. Now we apply this database to the MpSoC framework which generates the architecture and application mapping shown in Figure 7.8. We note that the synthesized architecture has total 4 processors, 2 instances each of the processor types PR2 and PR3. Here, some of channels are mapped to the local memories and rest of them got mapped to the shared memories.

Table 7.3 shows some of the experimental results when various architectural synthesis problem instances were applied to the MILP solver *LP Solve* [46]. This MILP solver uses branch and bound algorithms to solve the MILP problems. In these experiments, we observe that for small process networks (≤ 4 processes), the solver produces the solution very fast,

seed	0
jitt	0.5
nr_pi	3
ub_nr_po	2
lb_nr_processes	10
nr_processors	4
nr_memories	1
nr_switches	2
ub_nr_in_chns	3
ub_nr_out_chns	3
ub_nesting_level	3
channel_attribute	thr 3000 1000
channel_attribute	channel_size 4 2
channel_attribute	token_size 1000 200
channel_attribute	energy_per_byte 0.001 0
process_processor_attributes	n_cyc_avg 10000 6000
processor_attributes	freq 70 30
processor_attributes	contxt 200 0
processor_attributes	cost_proc 100000 50000
processor_attributes	cost_base_lm 20 5
memory_attributes	bwmm 20 4
memory_attributes	cost_base_sm 15 4
switch_attributes	cost_sw_in 2000 0

Figure 7.5: Attributes to generate database by RPNG

but as we keep on increasing the process network size, runtime taken by the MILP solver goes up significantly. Moreover for larger testcases, the solver is not able to produce the

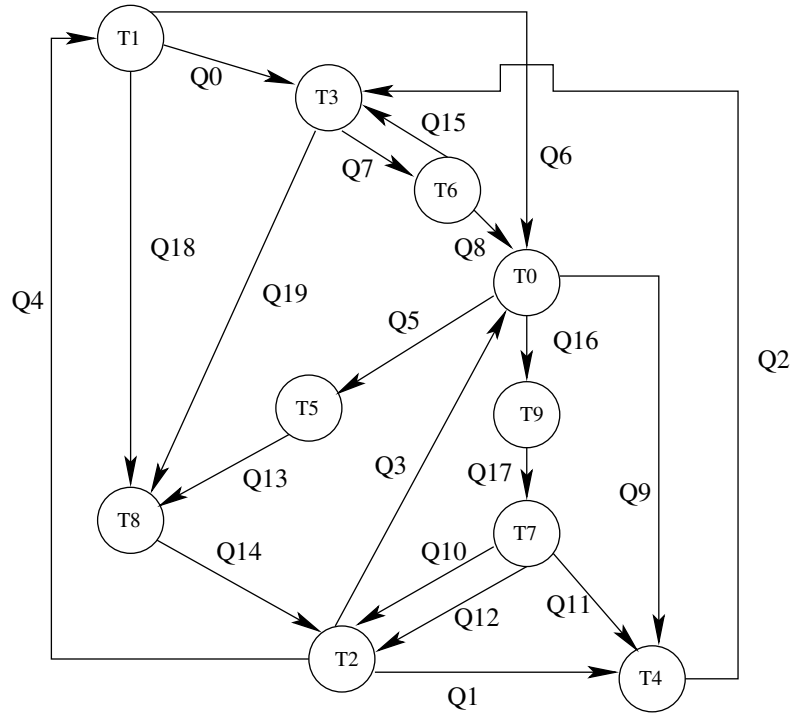


Figure 7.6: Process Network generated by RPNG using parameters of Figure 7.5

solution even after several hours. We did these experiments on a workstation having Intel XEON [79] CPU running at 2.20GHz and 1GB RAM.

Problem instance	Number of processes	Number of queues	Run time
1	4	6	< 1 second
2	4	8	< 1 second
3	5	19	< 1 second
4	5	15	< 1 second
5	6	15	8 seconds
6	6	21	25 minutes
7	7	20	no solution in 4 hours
8	9	27	no solution in 4 hours

Table 7.3: Runtime taken by MILP solver to solve various problem instances

```

@Processor 1 {
# freq  contxt  cost_proc  cost_base_lm
 51.436094  200.000000  146292.014112  15.673150

#-----
# process_id  n_cyc_avg
0  10395.899650
1  7493.718362
2  5523.994231
3  7515.170320
4  9886.916368
5  14066.491034
6  17167.959664
7  6332.840525
8  6832.687749
9  17399.381740
}

```

Figure 7.7: Partial database generated by RPNG for attributes of Figure 7.5

Table 7.4 shows some of the other experimental results. We performed these experiments to test the scalability of our MpSoC synthesis framework. The last row in this table gives the runtime taken to reach the solution for various problem instances. In these experiments, we note that as long as the total number of processes and channels are ≤ 480 , runtime to arrive at the synthesis solution is reasonable. When the process network size is such that there are ≥ 250 processes and ≥ 500 , runtime of the heuristic based framework goes into hours due to its $O(|Q|^4)$ complexity where $|Q|$ is the number of channels in the process network.

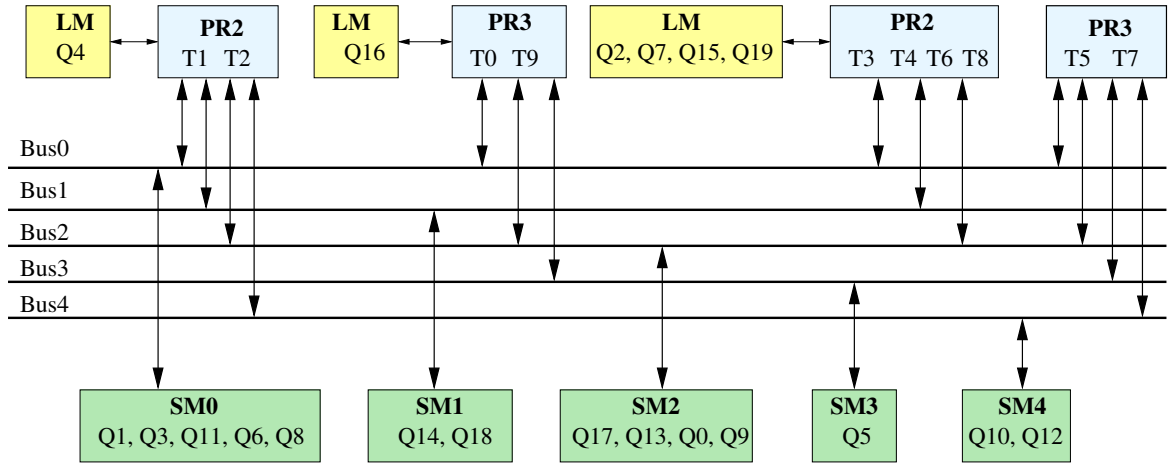


Figure 7.8: Architecture synthesized for process network of Figure 7.6

Problem instance	Number of processes	Number of queues	Run time
1	5	8	< 1 sec
2	10	21	< 1 sec
3	40	80	25 sec
4	80	160	90 sec
5	120	240	7 mins
6	160	320	28 mins
7	200	399	1 hour 25 mins
8	250	500	4 hours
9	300	600	10 hours

Table 7.4: Experiments to test scalability of MpSoC synthesis framework

7.3 Validation

Apart from doing experiments as explained above, we also compared quality of solution given by our approach presented in this thesis against solution provided by the mixed integer linear programming (MILP) formulation of the same problem. Table 7.5 shows

results of these experiments. We did these experiments on process networks which had number of processes ≤ 8 and number of channels ≤ 15 . We observed that for these cases, the solution given by the heuristic based framework was mostly close to that of MILP formulation, but sometimes it was poorer with the worst case being 16%. For validation, we could not obtain results for larger process networks (number of processes > 8 and number of channels > 15), because for these test cases, MILP solver took very large amount of time as shown in Table 7.3.

Problem instance	Number of processes	Number of queues	Solution cost by Heuristic / Solution cost by MILP	Run time MILP	Run time Heuristic
1	5	8	1.028	< 1 sec	< 1 sec
2	5	11	1.007	13 sec	< 1 sec
3	5	13	1.000	< 1 sec	< 1 sec
4	5	15	1.008	< 1 sec	< 1 sec
5	6	10	1.006	1 sec	1 sec
6	6	15	1.000	6 sec	1 sec
7	6	17	1.141	21 sec	1 sec
8	6	21	1.165	25 min	1 sec

Table 7.5: Quantitative comparison of synthesis problem solved by MILP solver vs. Heuristic based implementation

7.4 Summary

In this Chapter, we described various experimental results for MpSoC synthesis. These experiments show that the synthesis framework presented in Chapter 5 is quite scalable and produces solutions of reasonable quality quickly. These experiments also prove the usefulness of RPNG as an enabler for MpSoC synthesis research.

Chapter 8

Conclusions and Future Work

Process networks capture the parallelism present in the media applications quite well. We noticed that the problem of MpSoC architecture synthesis for applications modeled as process networks had not been adequately addressed in literature. So, we worked on this thesis with the objective to develop an integrated approach for automatic synthesis of optimal architectures for process networks. Towards this objective, we developed a framework which not only solves the MpSoC synthesis problem, but also allows quick validation. In this Chapter, we summarize our key contributions and discuss proposed future work.

8.1 Contributions

In this thesis, the problem of synthesis of multiprocessor SoCs for the process networks has been formulated and a methodology presented for the same. Our key contributions are as follows.

We formulated the MpSoC synthesis as two mixed integer linear programming problems. One can use the first MILP for allocation of processors, memories and mapping of application onto the architecture. The second MILP can be used to explore various interconnection networks. To get the overall minimum cost solution, these two MILPs can

be given to the MILP solver together. These MILPs can also be solved separately which may not give the overall minimum cost solution, but significantly simplifies the problem complexity.

We developed a new heuristic based framework in which the MpSoC synthesis problem can be solved for cost as well as power optimization. To perform resource allocation and application mapping, this framework works in two stages. In the first stage, the solution is generated using a dynamic programming algorithm for cost as well as power optimization and then, in the second stage, iterative refinements are done.

We developed analytical models to estimate contention in shared memory heterogeneous multiprocessors having non-uniform memory access. These models take into account the re-submissions of requests which are denied due to shared resource contention. Our estimation results show good correspondence to the simulation results. These estimation models aid in evaluating a particular mapping of the application onto the heterogeneous multiprocessor architecture during design space exploration.

We also developed a tool, RPNG, for randomly generating process networks. RPNG can be used for generating test cases for tools related to synthesis of application specific multiprocessor architectures. Apart from generating computation and communication characteristics of the process network in the database, RPNG also produces the source code which executes without deadlock. This code can also be used as a workload for simulation purposes to study the underlying architecture. By generating a number of process networks quickly, RPNG allows fast validation and enables research in MpSoC synthesis domain.

8.2 Future Work

There are a number of ways our work can be extended.

- In this thesis, we assumed that the sizes of local and shared memories are not fixed

and mapping algorithm is free to map channels of the process network anywhere so as to minimize cost. In practice, quite often, there is a constraint on the size of memories. So, the MpSoC synthesis framework needs to be extended to consider this constraint as well.

Adding memory size constraint in the architecture evaluation should not be difficult as this introduces only one new constraint.

- We considered buses and switches as the interconnection topologies. Other topologies such as the network of switches etc. should also be evaluated.

As part of evaluating a new interconnection topology, we also need to come up with corresponding estimation technique for the conflicts. Hence, this extension might not be straightforward.

- Many applications have soft real time performance constraints in the form of quality of service (QoS) requirements. The MpSoC synthesis framework considering QoS will be able to avoid over-design of the system.

Considering QoS in synthesis will introduce three sub-problems: distribution of overall QoS of the application to individual components of the application, finding correlation among processes and incorporating QoS requirements of individual processes into overall performance constraints. Hence, it appears that QoS consideration in synthesis will be non-trivial.

- After mapping the process network onto the architecture, the total context switch overhead during execution is the function of channel sizes of the process network. In this thesis, we assumed that channel sizes are pre-specified. Automatic determination of channel sizes would result in further memory optimizations. The synthesis framework should be extended to determine this parameter also.

Decision of channel sizes will introduce one more integer variable. This decision

doesn't appear to be straightforward because potentially there are a large number of choices. One possibility to make the problem simpler is to put range constraint on the channel sizes.

- In this thesis, we mainly focused on the resource allocation and application mapping problems and assumed that in the actual system implementation, a dynamic scheduler will take care of the run-time scheduling requirements. This scheduler will not only affect performance (by reducing number of context switches), but also affect energy usage. An interesting extension to our MpSoC framework would be to explore various dynamic scheduling policies while performing application mapping onto the architecture.

Since there are a large number of dynamic scheduling policies possible, choosing one of them will not be trivial. Essentially we require a framework in which we can evaluate various scheduling policies from performance and energy point of view. Further, this framework should also be generic enough to accommodate new scheduling policies for evaluation.

- It would be interesting to compare the fast approximate solutions produced by some of the MILP solvers with that of the proposed heuristics.

Studying solutions produced by heuristic based framework against approximate solutions provided by MILP should be straightforward and is subjected to the availability of such solver.

- Another interesting study would be to compare the automated solution against what is done in practice in terms of quality of solution and man-hours spent to finish the design. Typically, in practice, the system designer, based on his experience, incrementally refines the architecture and evaluates it by doing simulation. Since simulation is there in the design loop and designer also needs to put interfaces so that various parts

of the system work, comparing the automated solution against practice followed by the designer, is not straightforward and could be quite time consuming.

Bibliography

- [1] <http://www.cradle.com>.
- [2] <http://www.ti.com/omap2>.
- [3] <http://www.xilinx.com>.
- [4] <http://www.synopsys.com>.
- [5] Srijan: A Methodology for Synthesis of ASIP based Multiprocessor SoCs. Technical report, Department of Computer Science and Engineering, Indian Institute of Technology Delhi.
- [6] ARM. <http://www.arm.com>.
- [7] A. Baghdadi, D. Lyonnard, N. E. Zergainoh, and A. A. Jerraya. An Efficient Architecture Model for Systematic Design of Application-Specific Multiprocessor SoC. In *Proc. DATE*, 2001.
- [8] F. Balarin, P. D. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

-
- [9] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer Magazine*, 36(4):45–52, 2003.
- [10] T. Basten and J. Hoogerbrugge. Efficient Execution of Process Networks. In *Proc. Communicating Process Architectures - 2001*, pages 1–14. IOS Press, Amsterdam, the Netherlands, 2001.
- [11] R. A. Bergamaschi and W. R. Lee. Designing systems-on-chip using cores. In *Proc. DAC*, 2000.
- [12] L. N. Bhuyan, Q. Yang, and D. P. Agrawal. Performance of multiprocessor interconnection networks. *IEEE Computer*, 22(2):25–37, Feb. 1989.
- [13] E.-S. Chang, D. Gajski, and S. Narayan. An Optimal Clock Period Selection Method Based on Slack Minimization Criteria. 1(3):352–370, July 1996.
- [14] J.-M. Chang and M. Pedram. Codex-dp: Co-design of Communicating Systems Using Dynamic Programming. *IEEE Trans. on CAD*, 19(7):732–744, July 2000.
- [15] P. Chou, R. Ortega, and G. Borriello. The chinook hardware/software co-synthesis system. Technical Report 95-03-04, Dept. of Computer Science, University of Washington, Mar. 1994.
- [16] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Watanabe. Quasi-static Scheduling of Independent Tasks for Reactive Systems. *IEEE Transaction on Computer-Aided Design of Integrated Circuits*, 24(10):1492–1514, Oct. 2005.
- [17] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.
- [18] C. R. Das and L. N. Bhuyan. Bandwidth Availability of Multiple-Bus Multiprocessors. *IEEE Transaction on Computers*, C-34(10):918–926, Oct. 1985.

- [19] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: Hardware Software Cosynthesis of Heterogeneous Distributed Embedded Systems. *IEEE Trans. on VLSI Systems*, 7(1):92–104, Mar. 1999.
- [20] E. A. de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. International Symposium on System Synthesis (ISSS-2002)*, pages 68–73, Oct. 2002.
- [21] E. A. de Kock et al. YAPI: Application Modeling for Signal Processing Systems. In *Proc. Design Automation Conference (DAC-2000)*, pages 402–405, June 2000.
- [22] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task Graph Generation for Free. In *Proc. 6th International Workshop on Hardware/Software Codesign (CODES - 1998)*, pages 97–101, 1998.
- [23] B. K. Dwivedi, J. Hoogerbrugge, P. Stravers, and M. Balakrishnan. Exploring Design Space of Parallel Realizations: MPEG-2 Decoder Case Study. In *Proc. CODES*, 2001.
- [24] B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Automatic Synthesis of System on Chip Multiprocessor Architectures for Process Networks. In *Proc. Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2004)*, Stockholm, Sweden, pages 60–65, Sept. 2004.
- [25] B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Synthesis of Application Specific Multiprocessor Architectures for Process Networks. In *Proc. 17th International Conference on VLSI Design, Mumbai, India*, pages 780–783, Jan. 2004.
- [26] D. L. Eager, D. J. Sorin, and M. K. Vernon. AMVA Techniques for High Service Time Variability. In *Proc. SIGMETRICS*, pages 217–228. ACM, June 2000.
- [27] ESG. <http://embedded.cse.iitd.ernet.in/>.

-
- [28] M. J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett, 1996.
- [29] A. Fukuda. Equilibrium Point Analysis of Memory Interference in Multiprocessor Systems. *IEEE Transaction on Computers*, 37(5):585–593, May 1988.
- [30] J. Gaisler. LEON: A Sparc V8 Compliant Soft Uniprocessor Core. <http://www.gaisler.com/leon.html>.
- [31] O. P. Gangwal, A. Nieuwland, and P. Lippens. A Scalable and Flexible Data Synchronization Scheme for Embedded HW-SW Shared-Memory Systems. In *Proc. Int. Symposium on System Synthesis (ISSS-2001)*, pages 1–6, Oct. 2001.
- [32] F. E. Guibaly. Design and Analysis of Arbitration Protocols. *IEEE Transaction on Computers*, 38(2):161–171, 1989.
- [33] J. Gyllenhaal, B. Rau, and W. Hwu. HMDES version 2.0 specification, IMPACT, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-96-03, 1996.
- [34] M. A. Holliday and M. K. Vernon. Exact Performance Estimates for Multiprocessor Memory and Bus Interference. *IEEE Transaction on Computers*, C-36(1):76–85, Jan. 1987.
- [35] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw Hill, New York, 1984.
- [36] M. K. Jain, M. Balakrishnan, and A. Kumar. An Efficient Technique for Exploring Register File Size in ASIP Synthesis. In *Proc. International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES-2002)*, pages 252–261, Oct. 2002.

- [37] C. P. Joshi, M. Balakrishnan, and A. Kumar. A New Performance Evaluation Approach for System Level Design Space. In *15th International Symposium on System Synthesis (ISSS-2002)*, pages 180–185, Oct 2002.
- [38] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP Congress 74*. North Holland Publishing Co, 1974.
- [39] A. Kejariwal. Hardware estimation. B.Tech Thesis, Department of Computer Science & Engg., Indian Institute of Technology Delhi, New Delhi, India, 2002.
- [40] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, Dec. 1999.
- [41] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.
- [42] E. A. Lee. Overview of the Ptolemy Project. Technical Report UCB/ERL M03/25, University of California, Berkeley, July 2003.
- [43] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proc. of IEEE*, 83(5):773–801, May 1995.
- [44] J. Leijten, J. van Meerbergen, A. Timmer, and J. Jess. Stream Communication between Real-Time Tasks in a High-Performance Multiprocessor. In *Proc. Design, Automation and Test in Europe (DATE-1998)*, pages 125–131, 1998.
- [45] J. A. Leijten, J. L. van Meerbergen, A. A. Timmer, and J. A. G. Jess. PROPHID: A Heterogeneous Multi-Processor Architecture for Multimedia. In *Proc. International Conference on Computer Design (ICCD-1997)*, pages 164–169, 1997.
- [46] <http://sourceforge.net/projects/lpsolve>.

- [47] J. Luo and N. K. Jha. Low Power Distributed Embedded Systems: Dynamic Voltage Scaling and Synthesis. In *Proc. International Conference on High Performance Computing (HiPC-2002)*, pages 679–692, Dec. 2002.
- [48] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya. Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip. In *Proc. DAC*, 2001.
- [49] <http://www.eecs.harvard.edu/hube/research/machsuif.html>.
- [50] M. Madhukar, M. Leuze, and L. Dowdy. Petri-Net Model of a Dynamically Partitioned Multiprocessors System. In *Proc. Sixth International Workshop on Petri Nets and Performance Models*, pages 73–82, Oct. 1995.
- [51] J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen. LYCOS: the lyngby co-synthesis system. *Design Automation for Embedded Systems*, 2(2), 1997.
- [52] S. Mafeti, F. Gharsalli, F. Rousseau, and A. A. Jerraya. An optimal memory allocation for application-specific multiprocessor systems-on-chip. In *Proc. ISSS*, 2001.
- [53] M. A. Marsan and M. Gerla. Markov Models for Multiple Bus Multiprocessor Systems. *IEEE Transaction on Computers*, C-31(3):239–248, Mar. 1982.
- [54] Mibench. <http://www.eecs.umich.edu/mibench/>.
- [55] MPEG. *Information technology - Generic coding of moving pictures and associated audio information: Video*. ISO/IEC 13818-2, 1996.
- [56] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
- [57] A. Nandi and R. Marculescu. System-level power/performance analysis for embedded systems. In *Proc. DAC*, 2001.

- [58] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit for intel x86 architecture. In *Proc. ICCD*, 1996.
- [59] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proc. IEEE TCCA Newsletter*, Oct. 1997.
- [60] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, EECS Dept. Berkeley, 1995.
- [61] J. L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, 1977.
- [62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Universitat Darmstadt, Germany, 1962.
- [63] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and E. F. Deprettere. Exploring Embedded Systems Architectures with Artemis. *IEEE Computer*, (11):57–63, Nov. 2001.
- [64] R. R. Razouk. The use of petri nets for modeling pipelined processors. In *25th ACM/IEEE Design Automation Conference Proceedings*, pages 548–553, 1988.
- [65] K. V. Rompaey, D. Verkest, I. Bolsens, and H. D. Man. Coware A design environment for heterogeneous hardware/software systems. In *Proc. EuroDAC*, 1996.
- [66] D. Shin and J. Kim. Power-Aware Scheduling of Conditional Task Graphs in Real-Time Multiprocessor Systems. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED-2003)*, pages 408–413, Aug. 2003.
- [67] A. Sinha and A. P. Chandrakasan. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Design Automation Conference (DAC - 2001)*, pages 220–225, June 2001.

- [68] D. J. Sorin, J. L. Lemon, D. L. Eager, and M. K. Vernon. Analytic Evaluation of Shared-Memory Architectures. *IEEE Transaction on Parallel and Distributed Systems*, 14(2):166–180, Feb. 2003.
- [69] *The SPARC Architecture Manual Version 8*. <http://www.sparc.com/standards/v8.pdf>.
- [70] T. Stefanov and E. Deprettere. Deriving Process Networks from Weakly Dynamic Applications in System-Level Design. In *Proc. International Conference on HW/SW Codesign and System Synthesis (CODES+ISSS - 2003)*, pages 90–96, Oct. 2003.
- [71] STG. <http://www.kasahara.elec.waseda.ac.jp/schedule/>.
- [72] D. Sunada, D. Glasco, and M. Flynn. ABSS v2.0: a SPARC Simulator. In *Proc. SASIMI*, 1998.
- [73] The open systemC initiative. <http://www.systemc.org>.
- [74] D. F. Towsley. Approximate Models of Multiple Bus Multiprocessor Systems. *IEEE Transaction on Computers*, 35(3):220–228, Mar. 1986.
- [75] K. S. Vallerio and N. K. Jha. Task Graph Extraction for Embedded System Synthesis. In *Proc. 16th International Conference on VLSI Design (VLSI-2003)*, pages 480–485, Jan. 2003.
- [76] J. E. Veenstra and R. J. Fowler. MINT tutorial and user manual. Technical Report 452, Dept. of Computer Science, University of Rochester, New York, June 1993.
- [77] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: A Power -Performance Simulator for Interconnection Networks. In *Proc. 35th International Symposium on Microarchitecture (MICRO-35)*, pages 294–305, November 2002.
- [78] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hen-

- nessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.
- [79] XEON. <http://www.intel.com/products/server/processors>.
- [80] A. Yakovlev, L. Gomes, and L. Lavagno. *Hardware Design and Petri Nets*. Kluwer Academic Publishers, 2000.
- [81] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi, and A. A. Jerraya. A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design. In *Proc. CODES*, 2001.
- [82] W. M. Zuberek. Timed Petri Nets, Definitions, Properties, and Applications. *Microelectronics and Reliability*, 31(4):627–644, 1991.

Technical Biography of Author

Basant Kumar Dwivedi received B. E. in Electronics Engineering from MNNIT, Allahabad (India) in 1999 and M. Tech. in VLSI Design Tools and Technology from Indian Institute of Technology Delhi (India) in 2000. After completing his M. Tech., he joined Ph. D. in January 2001. During the course of his Ph. D., Basant had been working on "Synthesizing Application Specific Multiprocessor Architectures for Process Networks". Since November 2004, he has been working in Calypto Design Systems (I) Pvt. Ltd., Noida, India as a Senior Member Technical Staff.

List of Publications

1. B. K. Dwivedi, J. Hoogerbrugge, P. Stravers, and M. Balakrishnan. Exploring Design Space of Parallel Realizations: MPEG-2 Decoder Case Study. In *Proc. Int. Conf. on Hardware/Software Codesign, Copenhagen, Denmark*, pages 92–97, April 2001.
2. A. Singh, A. Chhabra, A. Gangwar, B. K. Dwivedi, M. Balakrishnan and A. Kumar. SoC Synthesis With Automatic Interface Generation. In *Proc. 16th International Conference on VLSI Design (VLSI-2003), New Delhi, India*, pages 585–590, Jan. 2003.
3. B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Synthesis of Application Specific Multiprocessor Architectures for Process Networks. In *Proc. 17th International Conference on VLSI Design, Mumbai, India*, pages 780–783, Jan. 2004.
4. B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Automatic Synthesis of System on Chip Multiprocessor Architectures for Process Networks. In *Proc. Int. Conf.*

on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2004), Stockholm, Sweden, pages 60–65, Sept. 2004.

5. B. K. Dwivedi, H. Dhand, M. Balakrishnan and A. Kumar. RPNG: A Tool for Random Process Network Generation. In *Proc. Asia and South Pacific International Conference in Embedded SoCs (ASPICES-2005), Bangalore, India, July, 2005.*
6. Basant K. Dwivedi, Arun Kejariwal, M. Balakrishnan, and Anshul Kumar. Rapid Resource-Constrained Hardware Performance Estimation. In *Proc. International Workshop on Rapid System Prototyping (RSP06), Chania, Crete, Greece, June 2006.*

