

## ABSTRACT

Title of Dissertation:      COMMUNICATION-DRIVEN CODESIGN  
FOR MULTIPROCESSOR SYSTEMS

Neal Kumar Bambha, Doctor of Philosophy, 2004

Dissertation directed by:    Professor Shuvra S. Bhattacharyya  
Department of Electrical and Computer Engineering

Several trends in technology have important implications for embedded systems of the future. One trend is the increasing density and number of transistors that can be placed on a chip. This allows designers to fit more functionality into smaller devices, and to place multiple processing cores on a single chip. Another trend is the increasing emphasis on low power designs. A third trend is the appearance of bottlenecks in embedded system designs due to the limitations of long electrical interconnects, and increasing use of optical interconnects to overcome these bottlenecks. These trends lead to rapidly increasing complexity in the design process, and the necessity to develop tools that automate the process. This thesis will present techniques and algorithms for developing such tools.

Automated techniques are especially important for multiprocessor designs. Programming such systems is difficult, and this is one reason why they are not as prevalent

today. In this thesis we explore techniques for automating and optimizing the process of mapping applications onto system architectures containing multiple processors. We examine different processor interconnection methods and topologies, and the design implications of different levels of connectivity between the processors.

Using optics, it is practical to construct processor interconnections having arbitrary topologies. This can offer advantages over regular interconnection topologies. However, existing scheduling techniques do not work in general for such arbitrarily connected systems. We present an algorithm that can be used to supplement existing scheduling techniques to enable their use with arbitrary interconnection patterns.

We use our scheduling techniques to explore the larger problem of synthesizing an optimal interconnection network for a problem or group of problems.

We examine the problem of optimizing synchronization costs in multiprocessor systems, and propose new architectures that reduce synchronization costs and permit efficient performance analysis.

All the trends listed above combine to add dimensions to the already vast design space for embedded systems. Optimizations in embedded system design invariably reduce to searching vast design spaces. We describe a new hybrid global/local framework that combines evolutionary algorithms with problem-specific local search and demonstrate that it is more efficient in searching these spaces.

COMMUNICATION-DRIVEN CODESIGN  
FOR MULTIPROCESSOR SYSTEMS

by

Neal Kumar Bambha

Dissertation submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
2004

Advisory Committee:

Professor Shuvra S. Bhattacharyya, Chairman/Advisor  
Professor Gang Qu  
Professor K. J. Ray Liu  
Professor Joe Mait  
Professor Jeff Hollingsworth

© Copyright by  
Neal Kumar Bambha  
2004

## DEDICATION

To my parents, and to Soraya, Valerie, and Michael

## ACKNOWLEDGEMENTS

I wish to thank my advisor, Professor Shuvra Bhattacharyya, for his guidance and encouragement in this endeavor. His enthusiasm for research and his insights into many different problems have motivated me greatly during the course of this work.

I would also like to acknowledge the support of the Army Research Laboratory, and to thank Dr. Paul Amirtharaj and Dr. Joe Mait for their encouragement.

It has been a pleasure to interact with many students in the DSPCAD group, including Nitin, Vida, Mukul, Ming-Yung, Bishnupriya, Ozkan, Ankush, and Mainak. Our collaborations and discussions have strengthened this thesis and enriched my experience here.

Finally, I would like to thank my family for their love, support, and patience during these years while I have focused on my studies.

## TABLE OF CONTENTS

<b>List of Figures</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Multiprocessor Embedded Systems . . . . .	2
1.2 Contributions of this Thesis . . . . .	4
1.2.1 Contention Analysis in Shared Bus Systems . . . . .	5
1.2.2 Architectures Designed for Optically Connected Systems . . . . .	6
1.2.3 Contention Analysis in Optically Connected Systems . . . . .	6
1.2.4 Scheduling for Arbitrarily Connected Systems . . . . .	6
1.2.5 Synthesizing an Optimal Interconnection Network . . . . .	7
1.2.6 Simulated Heating . . . . .	7
<b>2 Electronic Design Automation for Embedded Systems</b>	<b>9</b>
2.1 Dataflow . . . . .	9
2.2 Architectural Synthesis . . . . .	14
2.3 Scheduling . . . . .	19
2.4 Modeling Self-Timed Execution . . . . .	22
2.5 Interconnect Synthesis . . . . .	26
<b>3 System Architectures for Multiprocessor Embedded Systems</b>	<b>27</b>

3.1	Multiprocessor Program Execution Models . . . . .	28
3.2	Architectures Based on Dataflow . . . . .	30
3.3	Architectures Utilizing Optical Interconnects . . . . .	32
3.3.1	Optical Interconnect Technology . . . . .	32
3.3.2	Prototype Optically-Connected Systems . . . . .	35
3.4	Optically Connected System on Chip . . . . .	37
3.4.1	Global/Local Partitioning . . . . .	38
3.4.2	Typical Numbers . . . . .	42
3.5	Modeling Optically-Interconnected Systems with Synchronization Graphs	43
3.5.1	SLOT Architecture . . . . .	44
3.5.2	Dedicated Channel Fiber WDM Architecture . . . . .	46
3.5.3	One Wavelength Per Processor . . . . .	47
<b>4</b>	<b>Contention Analysis in Shared Bus Systems Utilizing the Period Graph</b>	<b>50</b>
4.1	Contention in Shared Bus Systems . . . . .	50
4.2	Constructing the Period Graph . . . . .	52
4.3	Fidelity of the Estimator . . . . .	58
4.4	Using the Period Graph in a Joint Power/Performance Algorithm . . . .	62
4.5	Genetic Algorithm Formulation . . . . .	63
4.6	Simulated Annealing Algorithm . . . . .	64
4.7	Results of Voltage Scaling using Period Graph . . . . .	65
4.8	Summary of Period Graph Work . . . . .	70
<b>5</b>	<b>Contention Analysis in Optically Connected Systems</b>	<b>71</b>
5.1	Ordered Transactions . . . . .	72
5.1.1	Ordered Transactions Concept . . . . .	72



5.1.2	Synchronization Constraints . . . . .	73
5.1.3	Ordered Transactions Graph . . . . .	75
5.2	WDM Ordered Transactions . . . . .	79
5.2.1	Optical Components . . . . .	84
5.2.2	Transaction Ordering . . . . .	86
5.2.3	Experiments . . . . .	90
<b>6</b>	<b>Scheduling for Arbitrarily Connected Systems</b>	<b>96</b>
6.1	Implications of Increased Connectivity . . . . .	97
6.1.1	Topology Graph . . . . .	98
6.1.2	Effect of Connectivity on a Simple Mapping Algorithm . . . . .	99
6.2	Connection Topologies . . . . .	99
6.3	Connectivity and Scheduling Flexibility . . . . .	105
6.4	Complexity of the Constraint Algorithm . . . . .	113
6.5	Incorporating Feasibility and Flexibility into Scheduling . . . . .	114
6.6	Scheduling Experiments using Flexibility . . . . .	115
6.6.1	Power Reduction with Single Hop Communication . . . . .	118
6.7	Summary of Flexibility Work . . . . .	119
<b>7</b>	<b>Synthesizing an Efficient Interconnect Network</b>	<b>120</b>
7.1	Greedy Interconnect Synthesis Algorithm . . . . .	122
7.1.1	Experiments with TPLA . . . . .	122
7.2	Link Synthesis using Genetic Algorithm . . . . .	124
7.2.1	Genetic Algorithm Overview . . . . .	124
7.2.2	Problem representation . . . . .	127
7.2.3	Fanout Constraints . . . . .	128

7.2.4	Crossover and Mutation Operators . . . . .	129
7.2.5	Experiments . . . . .	132
7.3	Using Graph Isomorphism . . . . .	134
<b>8</b>	<b>Design Space Exploration Using Simulated Heating</b>	<b>141</b>
8.1	Introduction . . . . .	141
8.1.1	PLSA for Voltage Scaling . . . . .	145
8.1.2	PLSA for Interconnect Synthesis . . . . .	150
8.1.3	PLSA for Ordered Transactions . . . . .	152
8.2	Hybrid Global/Local Search Related Work . . . . .	153
8.3	Simulated Heating . . . . .	157
8.3.1	Basic Principles . . . . .	158
8.3.2	Optimization Scenario . . . . .	159
8.4	Simulated Heating Schemes . . . . .	163
8.4.1	Static Heating . . . . .	163
8.4.2	Dynamic Heating . . . . .	166
<b>9</b>	<b>Simulated Heating Experiments</b>	<b>169</b>
9.1	Simulated Heating for Voltage Scaling . . . . .	169
9.1.1	Voltage Scaling Problem Statement . . . . .	170
9.1.2	GSA: Evolutionary Algorithm for Voltage Scaling . . . . .	170
9.2	Simulated Heating for Memory Cost Minimization . . . . .	172
9.2.1	Background . . . . .	172
9.2.2	MCMP Problem Statement . . . . .	173
9.2.3	Implementation Details for MCMP . . . . .	174
9.2.4	GSA: Evolutionary Algorithm for MCMP . . . . .	174

9.2.5	PLSA: Parameterized CDPPO for MCMP . . . . .	175
9.3	Simulated Heating for Binary Knapsack Problem . . . . .	176
9.3.1	Implementation . . . . .	177
9.4	Experiments . . . . .	178
9.4.1	PLSA Run-Time and Accuracy for Voltage Scaling and MCMP	180
9.4.2	Standard Hybrid Approach for Voltage Scaling and MCMP . . .	180
9.4.3	Static Heating Schemes for Voltage Scaling and MCMP . . . .	183
9.4.4	Dynamic Heating Schemes for Voltage Scaling and MCMP . . .	189
9.4.5	Knapsack PLSA Run-Time and Accuracy . . . . .	189
9.4.6	Knapsack Standard Hybrid Approach . . . . .	195
9.4.7	Knapsack Static Heating Schemes . . . . .	195
9.4.8	Knapsack Dynamic Heating Schemes . . . . .	195
9.5	Comparison of Heating Schemes . . . . .	199
9.5.1	Effect of Population Size . . . . .	201
9.6	Discussion . . . . .	206
9.7	Conclusions . . . . .	207
<b>10</b>	<b>Conclusions and Future Work</b>	<b>208</b>
	<b>Appendix</b>	<b>212</b>
.1	Random Graph Generation Algorithm . . . . .	212
	<b>Bibliography</b>	<b>216</b>

## LIST OF FIGURES

2.1	Marked Petri net. . . . .	12
2.2	Example of a problem graph, an architecture graph, and a chip graph. .	16
2.3	Specification graph corresponding to example of Figure 2.2. . . . .	16
2.4	Example of an application graph and an associated self-timed schedule.	20
2.5	Partition/overhead trade-off . . . . .	20
2.6	Example of IPC graph . . . . .	24
3.1	Shared-memory multiprocessor . . . . .	29
3.2	Distributed-memory multiprocessor . . . . .	30
3.3	Coarse-grain dataflow model . . . . .	33
3.4	FASTNet prototype. . . . .	36
3.5	Schematic side view of FASTNet . . . . .	39
3.6	Array of point source images . . . . .	40
3.7	Trade-offs in optical SoC . . . . .	41
3.8	Schematic of SLOT architecture. . . . .	45
3.9	Architecture for contention-free fiber-based SLOT. . . . .	47
3.10	Architecture for wavelength ordered transactions. . . . .	48
4.1	Schematic of a three processor shared bus architecture. . . . .	51
4.2	Pseudocode for constructing the period graph. . . . .	55

4.3	An illustration of the period graph construct. . . . .	56
4.4	Plot of fidelity of period graph estimator . . . . .	60
4.5	Average error of fidelity estimate . . . . .	61
4.6	Optimized power using period graph and simulation . . . . .	66
4.7	Optimized power using simulated annealing . . . . .	67
5.1	Pseudo-code for generating TPO graph. . . . .	76
5.2	$G_{TPO}$ after one and two passes. . . . .	77
5.3	Architecture for WDM OT . . . . .	81
5.4	Details of WDM OT . . . . .	82
5.5	WDMOT signals. . . . .	83
5.6	OADM using thin film filter . . . . .	85
5.7	WDM ordered transaction graph . . . . .	87
5.8	Function chooseCommunicationActor . . . . .	88
5.9	TPO heuristic [62]. . . . .	89
5.10	WDM ordered transactions algorithm. . . . .	90
5.11	Example of random graph. . . . .	91
5.12	Experiments with different ordering heuristics . . . . .	92
5.13	Experiments with different ordering heuristics . . . . .	94
5.14	WDMOT for varying number of processors. . . . .	95
6.1	Example of directional and directionless connectivity. . . . .	98
6.2	Impact of Connectivity on Search Efficiency. . . . .	100
6.3	Impact of Connectivity on Performance. . . . .	100
6.4	An irregular topology . . . . .	102
6.5	FFT1 application graph. . . . .	103

6.6	Single-hop topology . . . . .	103
6.7	Connectivity requirements of 100 benchmark applications. . . . .	104
6.8	Demonstrating flexibility metric . . . . .	106
6.9	Function $Rf^1(S)$ . . . . .	107
6.10	Function $Rb^1(S)$ . . . . .	107
6.11	Function bfsForward(). . . . .	108
6.12	Function bfsBackward(). . . . .	109
6.13	Feasibility function . . . . .	110
6.14	Makespan of FFT with and without flexibility metric . . . . .	116
7.1	Link synthesis using the TPLA algorithm. . . . .	123
7.2	Basic steps of a GA. . . . .	125
7.3	Crossover operator applied to array chromosome. . . . .	126
7.4	Incorrect crossover for link synthesis . . . . .	130
7.5	Correct crossover for link synthesis . . . . .	131
7.6	Comparison of TPLA and genetic algorithm for neural network appli- cation. . . . .	133
7.7	Example of two isomorphic graphs. . . . .	134
7.8	Isomorphically unique graphs $N = 4$ . . . . .	136
7.9	Enumeration of isomorphically unique graphs. . . . .	137
7.10	Deterministic link synthesis using isomorphism . . . . .	138
8.1	Hill climb local search for voltage scaling. . . . .	148
8.2	Monte Carlo local search for voltage scaling. . . . .	149
8.3	PLSA for interconnect synthesis. . . . .	151
8.4	PLSA for ordered transactions . . . . .	153

8.5	Simulated heating vs. traditional approach . . . . .	159
8.6	Global/Local Search Hybrid. . . . .	162
8.7	Different types of simulated heating . . . . .	164
9.1	Voltage scaling the period graph. . . . .	171
9.2	Pair swap pseudo-code. . . . .	179
9.3	Local search run times vs. $p$ . . . . .	181
9.4	MCMP results for standard hybrid. . . . .	182
9.5	MCMP number of generations vs. $p$ . . . . .	184
9.6	Results for standard hybrid using Monte Carlo. . . . .	185
9.7	Static heating for MCMP. . . . .	186
9.8	Static heating/Monte Carlo for FFT2. . . . .	188
9.9	Dynamic heating for MCMP. . . . .	190
9.10	Dynamic heating/Monte Carlo for FFT2. . . . .	191
9.11	Time spent in different parameter ranges. . . . .	192
9.12	Knapsack local search run time. . . . .	193
9.13	Error for standard hybrid approach. . . . .	194
9.14	Knapsack standard hybrid. . . . .	196
9.15	Knapsack static heating. . . . .	197
9.16	Dynamic heating binary knapsack . . . . .	198
9.17	Optimal value of $p$ . . . . .	199
9.18	Comparison of heating schemes for MCMP. . . . .	202
9.19	Comparison of heating schemes for voltage scaling. . . . .	203
9.20	Static heating with different population sizes. . . . .	204
9.21	Dynamic heating with different population sizes. . . . .	205

1	Random graph algorithm procedures. . . . .	213
2	Random graph algorithm procedures. . . . .	214
3	Random graph algorithm main. . . . .	215



# Chapter 1

## Introduction

The semiconductor industry has demonstrated remarkable progress during the past four decades. For society, this has meant a continual decrease in the cost of electronic devices, from computers to mobile phones to consumer electronics, and their increasing prevalence in our lives. Much of this progress results from the ability to exponentially decrease minimum feature sizes used to fabricate integrated circuits. The most frequently cited trend is Moore's Law, which states that the number of components on a chip doubles every 18 months. The International Technology Roadmap for Semiconductors predicts that by the year 2007, it will be possible to place 800 million transistors in a one square centimeter chip. At the same time, design cycle times have decreased, and interconnects between processing elements are becoming an increasing bottleneck. For a system designer, the biggest challenges involve making effective use of this huge potential functionality, and dealing with the associated complexity. In many ways, time is a much more precious commodity for designers today than is chip area. For this reason, tools that automate the design process are essential for the continued progress of the industry. There has been much research done on lower level design tools which optimize and produce a physical layout for a circuit that has been described in a sufficient

amount of detail. Less work has been done on tools for automating the higher levels of design. This thesis will address several issues relating to high level design automation, with a focus on embedded multiprocessor systems.

A central theme in this work is the effect of communication costs and resource contention across processors in the system. We develop techniques and algorithms to deal with these effects in systems whose complexity ranges from low cost shared bus systems to high performance multiprocessor systems utilizing optical interconnects. Communication and contention effects, along with the nature of the application, influence the type of interconnect that is most effective. We discuss different interconnection methods and present algorithms for finding an optimal interconnect topology.

All our optimization problems involve searching large, complex design spaces. Indeed, through our work with a diverse variety of complex optimization problems, we have developed unique insights on general methods for addressing such problems. We present a broadly-applicable framework, which has been derived from these insights, for searching complex design spaces, and we describe how our optimization problems can be solved using this framework.

## **1.1 Multiprocessor Embedded Systems**

An embedded system is a combination of computing hardware and software designed to perform a dedicated function. It is usually part of a larger system, such as the processor in a cell phone. By contrast, a general purpose computing system such as a personal computer is designed to perform many functions. Embedded systems typically offer much higher performance, lower power, and lower cost for their dedicated function than a general purpose system performing the same function. Examples of embedded

systems include consumer devices like MP3 players and cell phones, military radar and imaging systems, and processors for automotive engine control.

The processing elements of an embedded system perform two main tasks—control and data stream processing. The control functionality consists of choosing between modes of operation for the device, based on inputs and state information. For example, a simple controller chip on a microwave oven controls the power level and starts and stops the oven based on the keypad inputs. Data stream processing, or digital signal processing (DSP), is required in devices such as cell phones, which must sample data from the radio receiver and convert it into a digital data stream using algorithms which might decrypt the signal and correct for reception errors. In this thesis we will focus on developing tools that optimize the signal processing (DSP) functionality of a system. Processors with architectures that are optimized to provide very powerful digital signal processing functionality are inexpensive, readily available, and prevalent in modern devices.

Applications like video processing and automated target recognition are extremely computationally intensive, and require this processing to be performed in real time. One way to meet these requirements is to design very large scale integrated (VLSI) application-specific integrated circuits (ASIC) that are customized for the specific task. The main problem with this approach is the long design cycle, and the fact that the design is not flexible—if there are changes to the specifications, a new set of ASICs must be designed and tested. Programmable solutions, by contrast, allow changes to be made late in the design cycle by rewriting the software. The use of standard processing cores that have been verified for correctness eliminates much of the error-prone testing and debugging associated with ASIC design. However, it is often the case that a single, standard DSP chip cannot deliver the performance required from the application. In these cases, one attractive solution is to utilize multiple processors. Manufacturers today

are able to place several processors on a single die. As the transistor count continues to increase this becomes more cost effective, since it is less expensive to verify and test a number of smaller, standard processing elements than to test a larger, more complicated design. This will make multiprocessor design increasingly important in the future. One trade-off that comes with using multiple processors is that programming them is more complex, since it is necessary to deal with issues such as synchronization, deadlock, interconnect architecture, and interprocessor communication costs. Software tools are needed that allow the designer to specify an application at a high level, and that automate the details like synchronization and code generation. This thesis explores algorithms and techniques to develop such tools.

## **1.2 Contributions of this Thesis**

One major theme of this thesis is an analysis of the effect of resource contention in multiprocessor systems. We develop methods to analyze the effects of contention, architectures that are optimized to deal with these effects, and synthesis techniques and algorithms tailored to these architectures.

We consider a variety of systems with different cost/performance tradeoffs. Each successive level of hardware complexity reduces the effects of communication cost and resource contention, allows higher performance, and presents unique optimization challenges for the designer. We present techniques to deal with each of these challenges.

We begin with a shared electrical bus system, which is the simplest, lowest cost solution. The effects of contention are the most pronounced in these systems, and performance analysis is also the most complicated. We present a technique that makes analysis more efficient in these systems.

In order to reduce synchronization costs and improve predictability in these systems, researchers have previously developed an *ordered transaction* strategy that adds a hardware controller to the shared bus system [102] and have analyzed the effects of communication costs in these systems [62]. In this thesis we present a modification of this idea that utilizes optical fiber interconnects. This has the effect of dramatically reducing communication resource contention in the system.

The final, most complex architecture we consider is a multiprocessor system utilizing free space interconnects. This can eliminate communication resource contention entirely. One unique challenge for this system is to determine an optimal partitioning of the chip area between regions that are connected electrically and regions that are connected optically.

The optically connected systems offer the the ability to tailor the interconnection network optimally for a specific application. This opens up a vast new design space and poses several interesting challenges in scheduling and interconnect synthesis. We present new scheduling, interconnect synthesis, and optimization techniques to address these challenges.

### **1.2.1 Contention Analysis in Shared Bus Systems**

A critical challenge in synthesis techniques for iterative applications is the efficient analysis of performance in the presence of communication resource contention. To address this challenge for shared bus systems we introduce in Chapter 4 the concept of the period graph. The period graph is constructed from the output of a simulation of the system, with idle states included in the graph, and its maximum cycle mean is used to estimate overall system throughput. We analyze the fidelity of this estimator. As an example of the utility of the period graph, we demonstrate its use in a joint power/performance volt-

age scaling optimization solution. We quantify the speedup and optimization accuracy obtained using the period graph compared to using simulation only.

### **1.2.2 Architectures Designed for Optically Connected Systems**

In Chapter 3 we will discuss the role that optical interconnects can play in embedded multiprocessor systems, and derive some fundamental equations relating to optically connected systems on chip. We will introduce three architectures on which a broad class of high-throughput, self-timed DSP applications can be analyzed accurately using efficient graph-theoretic algorithms.

### **1.2.3 Contention Analysis in Optically Connected Systems**

Shared bus systems are appealing due to their simplicity and low cost. This is the primary driver for many embedded systems applications. However, a shared bus sometimes cannot meet the performance requirements for systems with significant interprocessor communication. In these cases, a designer may consider using a more expensive optical interconnect. In Chapter 5 we will explain how we modified the IPC graph model [102] and the synchronization graph model [18] to work with the optical architectures developed in Chapter 3.

### **1.2.4 Scheduling for Arbitrarily Connected Systems**

Optics provide the ability to construct highly connected and irregular networks that are streamlined for particular applications. Using these networks, it is possible to implement application mappings that allow flexible, single-hop communication patterns between processors, which has advantages for reduced system latency and power. This flexibil-

ity is particularly promising for embedded DSP applications, which are highly parallel and typically have tight constraints on latency and power consumption. In Chapter 6 we discuss the development of scheduling methods for optically connected embedded multiprocessors. We demonstrate that existing scheduling techniques will deadlock if communication is constrained by number of hops. We detail an efficient algorithm for avoiding this deadlock, and demonstrate its performance on several benchmark examples.

### **1.2.5 Synthesizing an Optimal Interconnection Network**

The freedom to optimize interconnection patterns opens up a vast design space, and thus the design of an optimal interconnect structure for a given application or set of applications is a significant challenge. In Chapter 7, we illustrate both probabilistic and deterministic interconnection synthesis algorithms. A key distinguishing feature to our interconnect synthesis algorithms is that they work in conjunction with a scheduling strategy—most existing interconnect synthesis algorithms assume a given schedule.

### **1.2.6 Simulated Heating**

All of the optimization problems we have considered, such as dynamic voltage scaling, scheduling, and interconnect synthesis, involve the search of vast design spaces. Most DSP optimization problems that arise in hardware-software co-design also involve searching large design spaces. For many of these problems, efficient *local search* algorithms exist for refining arbitrary points in the design space into better solutions. In Chapter 8 we introduce a novel approach, called simulated heating, for systematically integrating parameterized local search into global search algorithms. Using the framework of simulated heating, in Chapter 9 we investigate both static and dynamic strategies

for systematically managing the trade-off between local search accuracy and optimization effort for the voltage scaling application mentioned earlier, as well as a memory cost minimization problem and a more widely known optimization problem (binary knapsack). We also explain how simulated heating can be used in the transaction ordering optimization problem and the interconnect synthesis optimization problem. The application of simulated heating to these last two problems is the subject of future work.



## **Chapter 2**

# **Electronic Design Automation for Embedded Systems**

As mentioned earlier, the trend toward increasingly complex designs makes automated design tools very attractive. Ultimately, we would like a single tool that could start with an abstract, system-level design description and produce details of an optimized, hardware implementation. To reach this goal, we must have a suitable framework for describing the system at a high level of abstraction. Automated tools should be able to use this high level specification to generate the details of the design. This chapter will discuss the dataflow specification, and how it can be used for high level design.

## **2.1 Dataflow**

Dataflow graphs have proven to be a very useful specification for signal processing systems for several reasons. First, they support block-diagram based visual programming. Block diagrams (also called signal flow graphs or flow charts), are a versatile and important method for expressing DSP designs. Some of the most powerful DSP design tools use block diagrams as their primary design language. In these tools, the user de-

scribes a signal processing system by assembling a block diagram from a library of block functions, such as various types of filters. Examples of commercially available tools using dataflow and visual programming are the Signal Processing Worksystem from Cadence [14] and System Canvas from Angeles Design Systems [82].

A second strength of the dataflow specification is that it effectively exposes the parallelism in the application. It is difficult to compile programs written in imperative programming languages such as C on parallel architectures, since these languages are known to over-specify the control specification and the streaming specification. Parallel languages such as Universal Parallel C [22], are extensions of the serial languages intended to be compiled on parallel machines. However, these languages make certain assumptions about the hardware and are not applicable to a general architecture. They also require the programmer to explicitly handle lower-level details that we would like to avoid. The dataflow model imposes minimal data-dependency constraints in its specification, which allows the compiler to effectively detect parallelism.

A third advantage of the dataflow model is that in certain restricted forms it enables efficient algorithms for determining whether a program will deadlock, and whether it can be implemented in a finite amount of memory. This is not possible in more general computational models, as will be discussed later.

We will focus on applications that can be described by synchronous dataflow graphs (SDF) [69], and its various extensions such as boolean dataflow (BDF) [20]. In the SDF model, streams of data flow through a network of computational nodes. A program is represented as a directed *dataflow graph*. The vertices of this graph, called *actors*, represent computations and the edges represent FIFO buffers that queue the data. The data, represented by *tokens*, are passed from the output of one computation to the input of another. The numbers of tokens produced and consumed by each actor is fixed. The

programmer specifies the function performed at each node. The only constraints that are placed on order of evaluation come from the data dependences in the graph.

Delays on SDF edges represent initial tokens, and specify dependencies between iterations of the actors in iterative execution. For example, if tokens produced by the  $k$ th invocation of actor  $A$  are consumed by the  $(k + 2)$ th invocation of actor  $B$ , then the edge  $(A, B)$  contains two delays.

Actors can be of arbitrary complexity. In DSP design environments, they typically range in complexity from basic operations such as addition or subtraction to signal processing subsystems such as FFT units and adaptive filters.

We refer to an SDF representation of an application as an *application graph*. In this thesis we will mostly concentrate on a form of SDF called *homogeneous* SDF (HSDF) that is suitable for dataflow-based multiprocessor design tools since it exposes parallelism more thoroughly. In HSDF, each actor transfers a single token to/from each incident edge. General techniques for converting SDF graphs into HSDF form are developed in [69]. We represent a dataflow graph by an ordered pair  $(V, E)$ , where  $V$  is the set of actors and  $E$  is the set of edges. We refer to the source and sink actors of a dataflow edge  $e$  by  $\text{src}(e)$  and  $\text{snk}(e)$ , we denote the delay on  $e$  by  $\text{delay}(e)$ , and we can represent an edge  $e$  by the ordered pair  $(\text{src}(e), \text{snk}(e))$ . We say that  $e$  is an *output edge* of  $\text{src}(e)$ ;  $e$  is an *input edge* of  $\text{snk}(e)$ ; and  $e$  is *delay-less* if  $\text{delay}(e) = 0$ . The execution time or estimated execution time of an actor  $\nu$  is denoted  $t(\nu)$ .

Fundamental work related to the dataflow model was the work on *computational graphs* by Karp and Miller [59]. In this model, the computation is represented as a directed graph where nodes represent operations and edges represent queues of data. Karp and Miller proved that computation graphs with certain properties are *determinate*, which means that the sequence of data values produced by each node does not depend

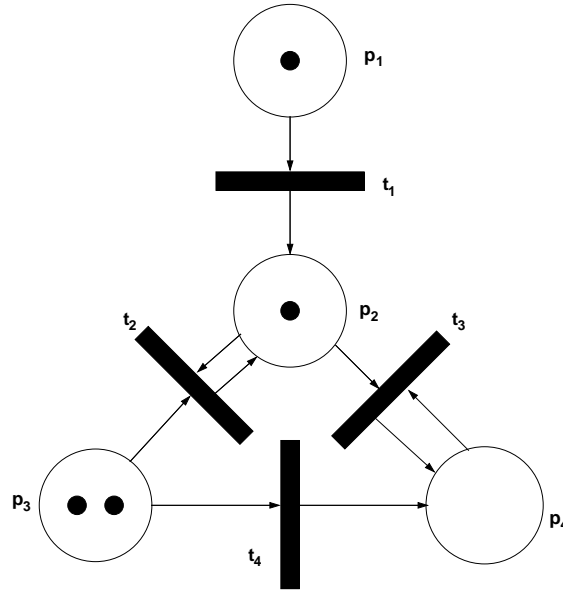


Figure 2.1: Marked Petri net.

on the *schedule*, or order of execution of the actors. They gave conditions to determine graphs whose computation can proceed indefinitely (avoidance of deadlock).

Several forms of dataflow are special cases of *Petri nets*. A general form of Petri nets is discussed in [86]. A Petri net is a directed graph,  $G = (V, A)$  where  $V = \{\nu_1, \dots, \nu_s\}$  is the set of vertices and  $A = \{a_1, \dots, a_r\}$  is a bag<sup>1</sup> of arcs. The set  $V$  can be partitioned into two disjoint sets  $P$ , representing *places* and  $T$ , representing *transitions*. Every arc in a Petri net connects a place to a transition or a transition to a place. Places may contain some number of tokens. A marking of a Petri net is a sequence of nonnegative integers for each place in the net, representing the number of tokens in the place. A Petri net together with a marking is called a *marked Petri net*. An example is given below in Figure 2.1. A Petri net executes by firing transitions. When a transition fires,

---

<sup>1</sup>A bag is distinguished from a set in that a given element can be included  $n$  times in a bag, so that the membership function is integer-valued rather than boolean-valued.

one token is removed from each input place of the transition and one token is added to each output place of the transition. A transition that has enough tokens on its input places to fire is *enabled*. Enabled transitions may fire, but are not required to. Firing may occur in any order and may continue as long as at least one transition is enabled. In Figure 2.1, transitions  $t_1$ ,  $t_2$ , and  $t_4$  are enabled. The marking can be represented as a vector  $\{1, 1, 2, 0\}$ . If transition  $t_4$  is fired, the new marking will be  $\{1, 1, 1, 1\}$  and transition  $t_3$  will be enabled.

*Marked graphs* are a subclass of Petri nets. A marked graph is a Petri net in which every place has exactly one input transition and one output transition. A marked graph can be represented by a graph with only a single type of node corresponding to transitions, with the data tokens considered to exist on the arcs. This representation is standard in dataflow. The properties of marked graphs were first investigated in [27].

The application of dataflow to computer architectures and programming languages was pioneered by Dennis [32]. The dataflow model of computer architecture was designed to enforce the ordering of instruction execution according to data dependencies. Execution of instructions is driven by the availability of data, as opposed to the more conventional von Neumann computer where the execution of instructions is controlled by a program counter. In a static dataflow machine, dataflow graphs are executed directly maintaining at the machine level a representation of the program as a dataflow graph and by providing hardware capabilities to detect when an actor has sufficient data to fire. There is a restriction that at most one data value can be queued on an edge at one time. This enables the storage for edges to be determined at compile time. However, this restriction also limits the amount of parallelism that can be extracted from loops. The *tagged-token dataflow model* [2, 47] was created to overcome this restriction. This model supports the execution of loop iterations and function invocations in

parallel. Data values are carried by tokens that include a three-part tag. The first part of the tag marks the current procedure invocation, the second part of the tag marks the loop iteration number, and the third part of the tag identifies the instruction number. Dataflow computers successfully address the problems of synchronization and memory latency, but are not as successful in coping with the resource requirements of large amounts of parallelism in the code. This is due to the overhead in keeping track of the data tags. Although some research continues on dataflow computers, none are in commercial development today. Most research into dataflow today applies to program representation.

Synchronous dataflow (SDF) is a restricted version of dataflow in which the number of tokens produced and consumed by an actor on each edge is fixed and known at compile time. Application of the SDF model to programming of multirate DSP systems was originated by Lee and Messerschmitt [69]. Lee and Messerschmitt provided efficient techniques to determine at compile time whether or not an arbitrary SDF graph has a periodic schedule that neither deadlocks nor requires unbounded buffer sizes. They also presented efficient methods for constructing such a periodic schedule whenever one exists. The SDF model has been successful at describing a large class of DSP applications and has been utilized in numerous design environments. Techniques for compiling general SDF programs for multirate DSP systems into efficient uniprocessor implementations that minimize both code and data memory requirements is presented in [15].

## **2.2 Architectural Synthesis**

System-level synthesis requires as a first step the selection of an architecture. In some cases, the designer is given a fixed platform, so the number of computing elements (processors, functional units, etc.) is fixed in advance. More commonly in embedded system

design, there is at least some flexibility to choose the number and types of processing elements and their interconnection. Even with a fixed platform, there can be choices between which tasks are performed by dedicated hardware units and which tasks are performed in software. Modern systems consist of an increasing number of these processing elements, each of which can be highly complex. The design may be realized on a single chip (*system on chip* or SoC), in a multichip design using multi-chip modules (mcm), or on separate circuit boards.

The system synthesis problem can be described formally by means of a specification graph [105], which is a graph  $G_S = (V_S, E_S)$  consisting of  $D$  dependence graphs  $G_i(V_i, E_i)$  for  $1 \leq i \leq D$  and a set of mapping edges  $E_M$ , where  $V_S = \bigcup_{i=1}^D V_i$ ,  $E_S = \bigcup_{i=1}^D E_i \cup E_M$ , and  $E_M = \bigcup_{i=1}^{D-1} E_{Mi}$ . Here,  $E_{Mi} \subseteq V_i \times V_{i+1}$  for  $1 \leq i < D$ .

The specification graph consists of layers of dependence graphs, each corresponding to a different level of abstraction. For example, an application graph describes the algorithm, an architecture graph describes the architecture, and a chip graph describes the physical components of the system. An edge in the specification graph between a task and a resource means that task can be implemented by that resource.

This can be better described by considering an example. The example in Figure 2.2 was taken from [105]. Figure 2.2a) depicts an application graph with four computational nodes and three communication nodes (shaded). The architecture, depicted in Figure 2.2b), consists of a RISC processor and two dedicated hardware modules. The hardware modules are connected to each other by a point-to-point bus, and to the RISC processor by a shared bus. The architecture graph corresponding to Figure 2.2b) is shown in Figure 2.2c). The physical implementation consists of two separate chips shown in Figure 2.2d) with a corresponding chip graph depicted in Figure 2.2e). The specification graph is shown in Figure 2.3. The edges  $E_{M1}$  and  $E_{M2}$  describe all possi-

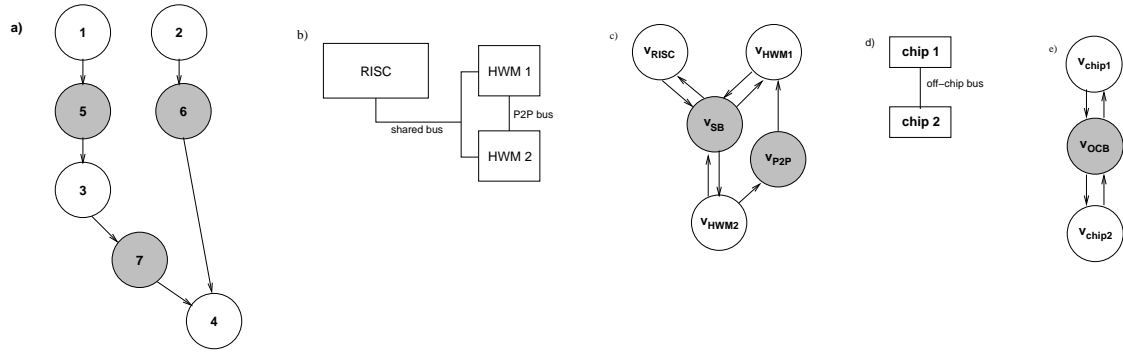


Figure 2.2: Example of a problem graph, an architecture graph, and a chip graph.

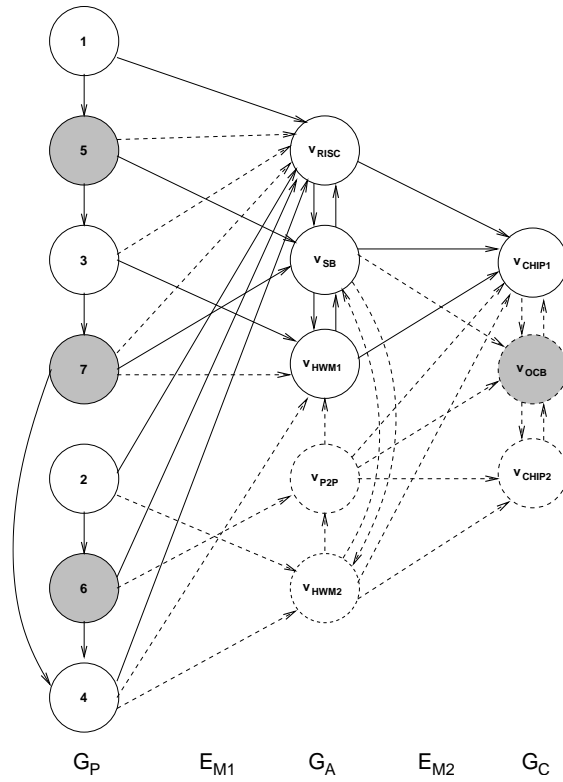


Figure 2.3: Specification graph corresponding to example of Figure 2.2.



ble mappings. The edges  $E_{M1}$  describe the possible mappings between the application graph and the architecture graph. We can see that task  $v_1$  can only be executed on  $v_{\text{RISC}}$  and task  $v_2$  can be executed on either  $v_{\text{RISC}}$  or  $v_{\text{HWM2}}$ . Communication task  $v_7$  can be executed on the shared bus  $v_{\text{SB}}$ . It can also be executed on  $v_{\text{RISC}}$  if tasks  $v_3$  and  $v_4$  both execute on  $v_{\text{RISC}}$ , or on  $v_{\text{HWM1}}$  if  $v_3$  and  $v_4$  also execute on  $v_{\text{HWM1}}$ . The edges  $E_{M2}$  describe the possible mappings between the architecture graph and the chip graph. From these edges we can see that any of the tasks in the architecture graph (the RISC processor, shared bus, point-to-point bus, and both hardware modules) can be implemented inside CHIP1, and that the shared bus  $v_{\text{SB}}$  can be handled by CHIP1 or by the off chip bus  $v_{\text{OCB}}$ . The dashed nodes and edges in Figure 2.3 are not *allocated* in the implementation. The specification graph allows us to state a formal definition for allocation, binding, and scheduling.

The **activation** of a specification graph  $G_S(V_S, E_S)$  is a function  $a : V_S \cup E_S \mapsto \{0, 1\}$  that assigns to each edge  $e \in E_S$  and each node  $v \in V_S$  the value 1 (activated) or 0 (not activated).

An **allocation**  $\alpha$  of a specification graph is the subset of all activated nodes and edges of the dependence graphs  $\alpha = \alpha_V \cup \alpha_E$  with  $\alpha_V = \{v \in V_S \mid a(v) = 1\}$  and  $\alpha_E = \bigcup_{i=1}^D \{e \in E_i \mid a(e) = 1\}$ . For the example above, the allocation of nodes is  $\alpha_V = \{v_{\text{RISC}}, v_{\text{HWM1}}, v_{\text{SB}}, v_{\text{CHIP1}}\}$ .

A **binding**  $\beta$  is the subset of all activated mapping edges so that  $\beta = \{e \in E_M \mid a(e) = 1\}$ . For the example above, the binding is

$$\begin{aligned} \beta = \{ & (v_1, v_{\text{RISC}}), (v_2, v_{\text{RISC}}), (v_3, v_{\text{HWM1}}), (v_4, v_{\text{RISC}}), (v_5, v_{\text{SB}}), (v_6, v_{\text{RISC}}), \\ & (v_7, v_{\text{SB}}), (v_{\text{RISC}}, v_{\text{CHIP1}}), (v_{\text{SB}}, v_{\text{CHIP1}}), (v_{\text{HWM1}}, v_{\text{CHIP1}}) \} \end{aligned}$$

so that all the architecture components are bound to CHIP1.

A **feasible binding**  $\beta$  is a binding that satisfies the following criteria:

1. Each activated edge  $e \in \beta$  starts and ends at an activated node.
2. For each activated node  $v \in \alpha_V$  with  $v \in V_i$ ,  $1 \leq i < D$  exactly one outgoing edge  $e \in V_M$  is activated.
3. For each activated edge  $e = (v_i, v_j) \in \alpha_E$  with  $e \in E_i$ ,  $1 \leq i < D$  either both operations are mapped onto the same node or there exists an activated edge  $\tilde{e} = (\tilde{v}_i, \tilde{v}_j) \in \alpha_E$  with  $\tilde{e} \in E_{i+1}$  to handle the communication associated with edge  $e$ , i.e.  $(\tilde{v}_i, \tilde{v}_j) \in \alpha_E$  with  $(v_i, \tilde{v}_i), (v_j, \tilde{v}_j) \in \beta$ .

It has been shown that the problem of finding a feasible binding is NP-complete [19].

A **schedule** is a function  $\tau : V_P \mapsto \mathbb{Z}^+$  that satisfies for all edges  $e = (v_i, v_j) \in E_P$  the condition  $\tau(v_j) \geq \tau(v_i) + \text{delay}(v_i, \beta)$  where  $\text{delay}(v, \beta)$  is the execution time delay of node  $v$  given a binding  $\beta$ . For the example above a valid schedule is  $\tau(v_1) = 0$ ,  $\tau(v_2) = 1$ ,  $\tau(v_3) = 2$ ,  $\tau(v_4) = 21$ ,  $\tau(v_5) = 1$ ,  $\tau(v_6) = 21$ ,  $\tau(v_7) = 4$ .

A valid **implementation** is a triple  $(\alpha, \beta, \tau)$  where  $\alpha$  is an allocation,  $\beta$  is a binding, and  $\tau$  is a schedule.

Finally, with the definitions above we can state the problem formally: **system synthesis** consists of minimizing a function  $h(\alpha, \beta, \tau)$  which describes an optimization goal, subject to

- $\alpha$  is a feasible allocation,
- $\beta$  is a feasible binding,
- $\tau$  is a schedule.

## 2.3 Scheduling

Implementing an algorithm specified as a dataflow graph (DFG) on a multiprocessor system requires “scheduling” the actors. Scheduling was defined formally in Section 2.2. Scheduling involves the tasks of (1) assigning actors in the DFG to processors, (2) ordering the execution of these actors on each processor, and (3) determining the start times of all the actors while maintaining the data precedence constraints of the DFG. Scheduling has been studied extensively in many contexts, and has been classified based on which of the tasks listed above are performed at compile time and which at run time [68].

If all three are performed at compile time, the scheduling strategy is said to be *fully static*. This method requires the least possible runtime overhead. The exact execution times of all the actors are assumed to be given in advance. The processors can run in lock step according to the schedule, and no explicit synchronization is required when they communicate data. However, the exact run times of the actors cannot usually be determined in advance, so the fully static strategy is often not practical.

For DSP applications, it is usually realistic to assume that good estimates for the execution times can be determined. Given this assumption, a *self-timed* [68] scheduling strategy can be employed, where the ordering of the actors on each processor is specified, but not the exact start times. Each processor waits for the data needed by an actor before executing that actor. This requires that the processors perform some run-time synchronization when they exchange data, so the run-time overhead is greater for this scheduling strategy. Examples of an application graph and a corresponding self-timed schedule are illustrated in Figure 2.4.

Another consideration in scheduling is the size or granularity of an actor. Figure 2.5 shows a trade-off between parallelism and communication overhead in a heterogeneous DSP system as the size of the actor is varied. It is repeated from the study by Sarkar [93].

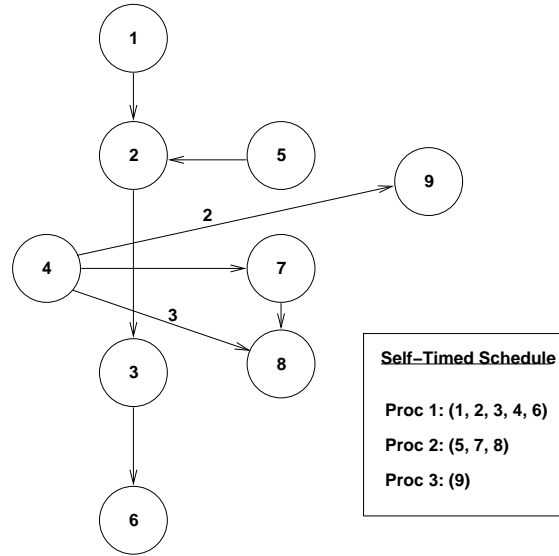


Figure 2.4: An example of an application graph and an associated self-timed schedule. The numbers on edges (4,8) and (4,9) denote nonzero delays.

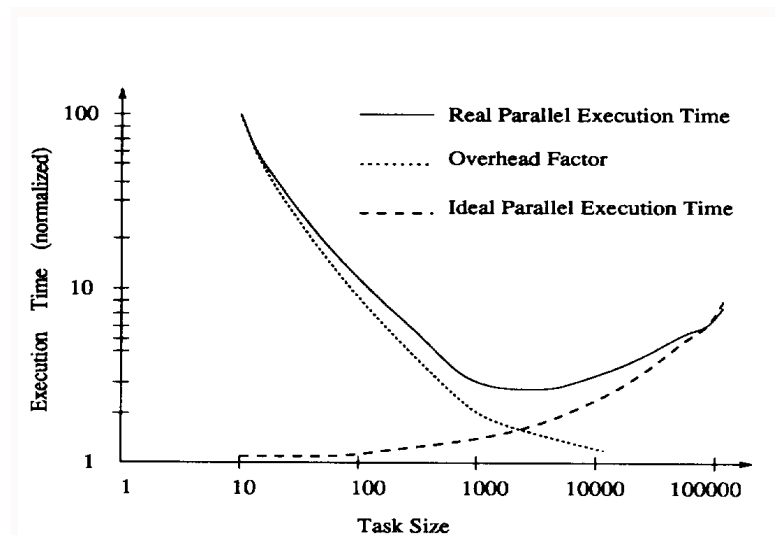


Figure 2.5: Partition-overhead trade-off [93].

The vertical axis is a measure of performance. As the average actor size is increased, the interprocessor communication (IPC) overhead drops. At the same time, there is a loss of parallelism, so the execution time for an ideal parallel system (with no IPC) increases. Partitioning algorithms try to find the optimal balance between these two factors. Sarkar developed a two-phase scheduling method. The first phase involved scheduling the input graph onto an ideal architecture in which there are no resource constraints or communication costs. This **infinite-resource multiprocessor architecture (IRMA)** consists of an infinite number of processors that are interconnected by a fully-connected crossbar interconnect (an interconnect in which every processor is directly connected to every other processor). The communication in the IRMA architecture is assumed to be simultaneous. In the second phase, the schedule derived for the IRMA architecture is modified to work on the resource-constrained target architecture.

For a system with fixed resource constraints, the multiprocessor partitioning and scheduling problems are NP hard [42], so heuristics must be used. Many such heuristics have been developed. Most existing scheduling heuristics try to minimize the schedule makespan, which is the time it takes for all the tasks to finish the first iteration (execution of one schedule period). However, most DSP applications are non-terminating; an example of a filter operating on an unbounded stream of speech samples. In this case, it is more appropriate to generate schedules that maximize the throughput. Scheduling heuristics can be classified into the following categories: list scheduling heuristics, graph decomposition heuristics, and critical path heuristics.

The most well-studied area in scheduling involves heuristics based on the idea of *priority lists* [31]. These heuristics use a priority list to define an ordering of the nodes in the graph, and use an algorithm that selects each function in order of priority for scheduling on an appropriate resource. In order to compute the priorities, the allocation

and binding steps described in section 2.2 need to be performed in advance.

For DFGs with edge weights and node weights, a path weight can be defined as the sum of the weights of both nodes and edges on the path. A critical path from a source node to a sink node is a path with maximal weight. In the critical path techniques, the graph is partitioned after examining the current critical path, zeroing an edge by combining the incident nodes into a cluster, and repeating the process on the new critical path. In the dominant sequence clustering algorithm by Yang and Gerasoulis [109], the decision to zero an edge is based on the new start time of the node at the beginning of the dominant sequence (the critical path after zeroing of one or more edges) and the start time of an unscheduled node most likely to be affected by the zeroing decision. If either of these start times is increased, the zeroing is not done. Due to the relative simplicity of the zeroing criteria, the complexity of this method is  $O((\nu + e) \log \nu)$ . The modified critical path algorithm by Wu and Gajski [108] considers as-late-as-possible binding, which is found by traversing the graph from the sink nodes to the source nodes and assigning the latest possible start time to each node. A node on the critical path is selected and placed on a different processor. The complexity of this method is  $O(\nu^2 \log \nu)$ .

## 2.4 Modeling Self-Timed Execution

In relation to the scheduling taxonomy of Lee and Ha [68], in this thesis we focus on the *self-timed* strategy and variations of the closely-related *ordered transactions* strategy optimized for optically-connected multiprocessors. The self-timed and ordered transaction strategies are popular and efficient for the DSP domain due to their combination of robustness, predictability, and flexibility [101]. In self-timed scheduling, each processor executes the tasks assigned to it in a fixed order that is specified at compile

time. Before executing an actor, a processor waits for the data needed by that actor to become available. Thus, processors are required to perform run-time synchronization when they communicate data. This provides robustness when the execution times of tasks are not known precisely or when they may exhibit occasional deviations from their compile-time estimates. Examples of an application graph and a corresponding self-timed schedule are shown in Figure 2.4.

The *ordered transaction* method is similar to the self-timed method, but it also adds the constraint that a linear ordering of the communication actors is determined at compile time, and enforced at run-time [102]. The linear ordering imposed is called the *transaction order* of the associated multiprocessor implementation. The transaction order, which is enforced by special hardware, obviates run-time synchronization and bus arbitration, and also enhances predictability. Also, if constructed carefully, it can in general lead to a more efficient pattern of actor/communication operations compared to an equivalent self-timed implementation [62].

Next we will examine two related graph-theoretic models, the *interprocessor communication graph (IPC graph)*  $G_{IPC}$  [101, 102] and the *synchronization graph*  $G_s$  [102], that are used to model the self-timed execution of a given parallel schedule for a dataflow graph. Given a self-timed multiprocessor schedule for  $G$ , we derive  $G_{IPC}$  by instantiating a vertex for each task, connecting an edge from each task to the task that succeeds it on the same processor, and adding an edge that has unit delay from the last task on each processor to the first task on the same processor. Also, for each edge  $(x, y)$  in  $G$  that connects tasks that execute on different processors, an IPC edge is instantiated in  $G_{IPC}$  from  $x$  to  $y$ . Figure 2.6 shows the IPC graph that corresponds to the application graph and self-timed schedule of 2.4. In this graph, the nodes labeled with “s” are nodes that send data and the nodes labeled with “r” are nodes that receive data. The numbers in

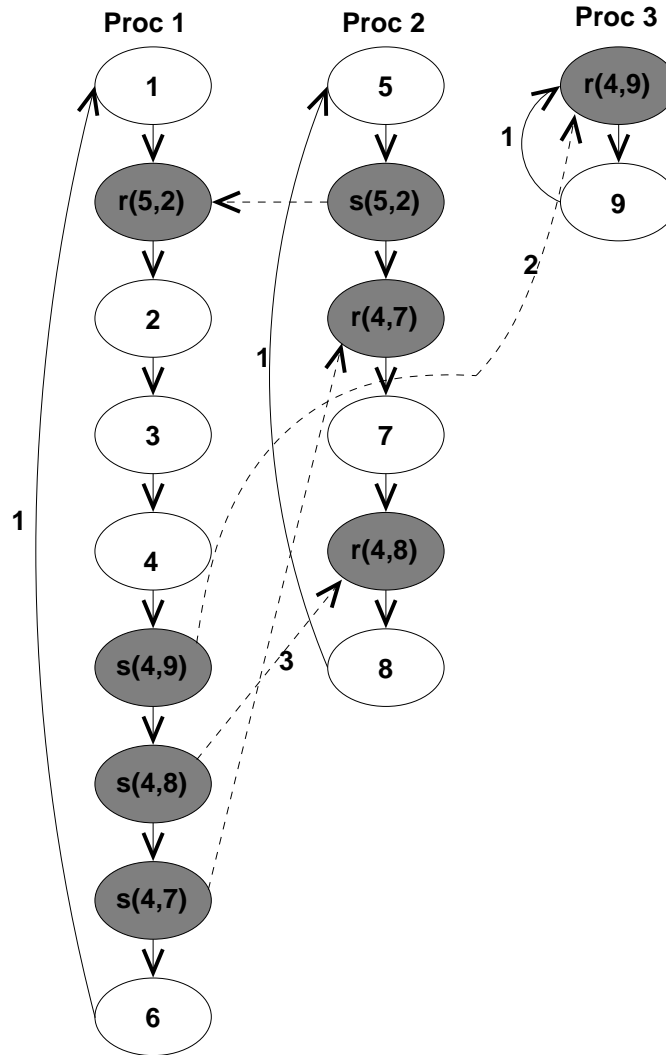


Figure 2.6: The IPC graph constructed from the application graph and schedule of Figure 2.4. Dashed edges represent IPC edges and shaded actors are communication actors (send and receive actors) that perform interprocessor communication. Numbers next to edges represent delays.



parentheses represent the sending and receiving actors. For example  $s(5, 2)$  represents a communication actor with data from actor 5 being sent to actor 2.

The non-communication vertices in  $G_s$  and  $G_{IPC}$  correspond to individual tasks of the application being implemented. Each edge in  $G_{IPC}$  and  $G_s$  is either an *intraprocessor edge* or an *interprocessor edge*. Intraprocessor edges model the ordering (specified by the given parallel schedule) of tasks assigned to the same processor. Interprocessor edges in  $G_{IPC}$ , called *IPC edges*, connect tasks assigned to distinct processors that must communicate for the purposes of data transfer, and interprocessor edges in  $G_s$ , called *synchronization edges*, connect tasks assigned to distinct processors that must communicate for synchronization purposes. We will discuss the synchronization graph in more detail in Chapter 5.

Each edge  $(v_j, v_i)$  in  $G_{IPC}$  represents the *synchronization constraint*

$$\text{start}(v_i, k) \geq \text{end}(v_j, k - \text{delay}((v_j, v_i))) \quad \forall k, \quad (2.1)$$

where  $\text{start}(v, k)$  and  $\text{end}(v, k)$  respectively represent the time at which invocation  $k$  of actor  $v$  begins execution and completes execution, and  $\text{delay}(e)$  represents the delay associated with edge  $e$ .

The IPC graph is an instance of Reiter's *computation graph* model [90], also known as the *timed marked graph* model in Petri net theory [86], and from the theory of such graphs, it is well known that in the ideal case of unlimited bus bandwidth, the average iteration period for the as-soon-as-possible (ASAP) execution of an IPC is given by the *maximum cycle mean (MCM)* of  $G_{IPC}$ , which is defined by

$$\text{MCM}(G_{IPC}) = \max_{\text{cycle } C \text{ in } G_{IPC}} \left\{ \frac{\sum_{v \in C} t(v)}{\text{Delay}(C)} \right\}. \quad (2.2)$$

The MCM can be computed efficiently—Karp's algorithm [58] runs in  $\Phi(nm)$  time where  $n$  is the number of actors in the graph and  $m$  is the number of edges. Dasdan and

Gupta [29] describe an algorithm based on Karp's algorithm that runs in (worst case)  $O(nm)$ , and always faster than Karp's algorithm.

## 2.5 Interconnect Synthesis

SoC design is moving toward a paradigm where reusable components called IP (for intellectual property) from different vendors can be combined to rapidly create a design. IP interface standards are being developed which define the services one IP component (or *IP core*) is capable of delivering, and which enable IP cores to work with on-chip buses and other interconnection networks. The SoC designer's task is then to choose the appropriate IP cores, map the application tasks onto these cores, and to construct a communication network and corresponding glue logic to connect these IP cores.

Interconnect synthesis is becoming an increasingly important part of system-level synthesis, given the larger number of blocks that must be interconnected and the increasing importance of interconnect delay to overall performance. To date, shared bus has been the dominant interconnect. However, researchers are now exploring a richer set of interconnection schemes, including crossbars, meshes, and other point-to-point topologies. We will explore interconnect synthesis in detail in Chapter 7.

## **Chapter 3**

# **System Architectures for Multiprocessor Embedded Systems**

There has been substantial research work in the areas of multiprocessor hardware and software for high performance, general purpose computing. These machines tend to be big and expensive, and are targeted toward solving large computational problems such as climate simulation. As mentioned in the Introduction, embedded systems can also utilize multiprocessor architectures, and some research work has focused on developing application-specific multiprocessor systems. Since these systems only need to support a limited number of programs, it is often possible to streamline the hardware architecture. We will focus on systems running applications that can be described by dataflow graphs. In these applications, parallelism is easier to identify and exploit because much more is known about the structure of the computation.

We will discuss the role that optical interconnects can play in embedded multiprocessor systems, and derive some fundamental equations relating to optically connected systems on chip. We will introduce three architectures on which a broad class of high-throughput, self-timed DSP applications can be analyzed accurately using efficient graph-theoretic algorithms.

### 3.1 Multiprocessor Program Execution Models

For sequential computers, the principal execution model in use today is the *von Neumann model* which consists of a sequential process running in a linear address space [45]. In 1966, Flynn [41] proposed a simple model of categorizing multiprocessors using this execution model as either Single Instruction Multiple Data (SIMD) or Multiple Instruction Multiple Data (MIMD) according to how they partition control and data among different processing elements. In a SIMD machine the same instruction is executed by multiple processors using different data streams. Each processor has its own data memory, but there is a single instruction memory and control processor. In a MIMD machine, each processor fetches its own instructions and operates on its own data. Using this terminology, we would call a uniprocessor a single instruction, single data stream (SISD) machine. MIMD machines fall into two categories—centralized shared-memory architectures and distributed memory architectures. Figure 3.1 [85] depicts the basic structure of a centralized shared-memory multiprocessor, where the processors and memory are connected by a shared bus. Processors communicate by writing and reading from locations in memory. In order to reduce the memory bandwidth requirement of the processors, memory cache is used. We may classify the data in the multiprocessor as *private data* if it is only used by a single processor, or *shared data* if it is used by multiple processors. The communication mechanism utilizes shared data. When data is migrated into a processor's cache, the bus bandwidth is reduced since this processor does not need to access main memory to fetch the data. Also, memory access time to cache is faster than to main memory. When the data is private data, the program execution is not affected. However, when shared data are cached, the data may be stored in multiple caches. This complicates the program execution, since there must be some way to reconcile the different copies of the data. This problem is called *cache coherence*, and

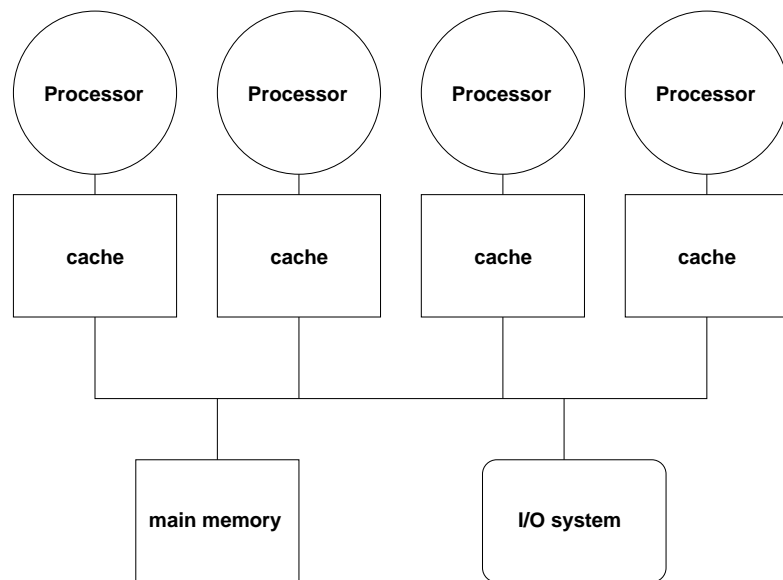


Figure 3.1: Basic structure of a centralized shared-memory multiprocessor. Multiple processor-cache subsystems share the same physical memory, typically connected by a bus.

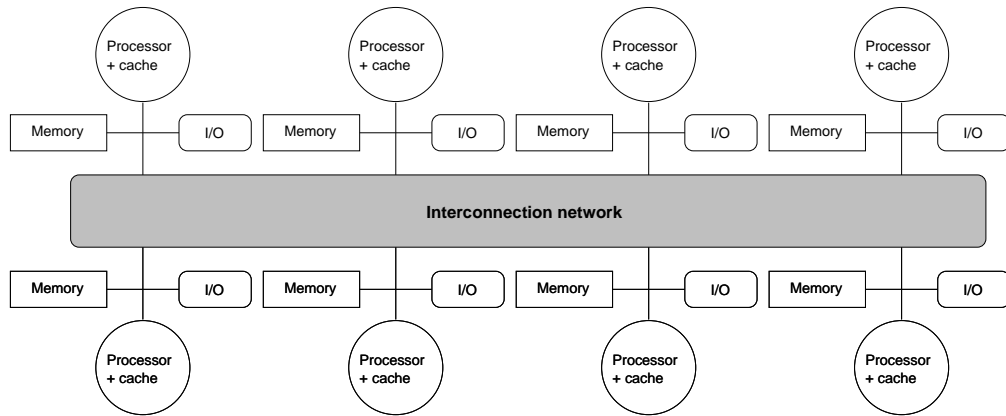


Figure 3.2: Basic structure of a distributed-memory multiprocessor. Individual nodes contain a processor, some memory, and an interface to an interconnection network that connects all the nodes. Individual nodes may themselves contain a small number of processors interconnected via a bus or other interconnect which is often less scalable than the global interconnection network.

has been well studied in general purpose computing [1]. For some embedded systems applications the cache is eliminated in order to reduce complexity and cost.

Figure 3.2 [85] depicts a distributed-memory machine, which has a physically distributed memory. These machines typically have larger processor counts, where a shared bus cannot handle the required communication bandwidth. Distributing the memory reduces the latency for access to the local memory. Compared to the shared-memory architecture, communication between processors is more complex.

## 3.2 Architectures Based on Dataflow

In the von Neumann architecture, all the data, the locations of the data, and the operations to be performed on the data, must travel between memory and CPU a word at a

time. This has been termed the “von Neumann bottleneck” [3]. Hardware architectures based on dataflow have been studied in order to avoid this bottleneck. The dataflow model of computation was discussed in Chapter 2. Dataflow models use dataflow program graphs to represent the flow of data and control. In this model an instruction may be executed (or *fired*) as soon as all its input operands are available. When an instruction fires, it consumes its input values and generates some output values. Because of this, the dataflow model is *asynchronous*. In a dataflow architecture the program execution involves receiving, processing, and sending out tokens containing data and a tag. Dependencies between data are translated into tag matching and transformation. Processing occurs when a set of matched tokens arrives at the execution unit. The matching unit and execution unit are connected by queues. Several types of architectures based purely on dataflow have been studied in the past. They differ in how the tokens are handled.

The *single token per arc* dataflow architecture was proposed by Dennis [34]. In this architecture, a dataflow graph is represented by a number of *activity templates*, each containing an instruction and *operand slots* for holding operand values. Only one token is allowed at a time on an arc. Acknowledge signals are used to enforce the single token rule, making it relatively simple to detect when a node is enabled. The *MIT Static Dataflow Architecture* [33] was a direct implementation of this model. One disadvantage of this architecture is that consecutive iterations of a loop can only partially overlap in time. Another is the additional token traffic caused by the acknowledgment tokens.

The *tagged-token dataflow* model [107] was created to allow loop iterations to proceed in parallel. In this model, each token contains a tag that defines the context in which the data value will be used. Multiple tokens are allowed on an arc. A node is enabled as soon as tokens with identical tags are present on each of its input arcs. Several groups produced prototype implementations of this model [13, 51, 104]. A disadvantage of this

model is that it is difficult to implement an efficient unit to handle the overhead of token matching.

It was found that computers implementing pure dataflow performed poorly on sequential code. This is due in part to the fine granularity—tasks correspond to operations such as a simple multiply or compare, and the overhead associated with token matching of these tasks. One solution to this problem is to combine a dataflow architecture with a von Neumann architecture. Most recent dataflow architectures employ a *coarse-grain* model in which the computational task of the dataflow actors consist of a number of instructions; the computation inside each task is executed on a von Neumann processor (often a commercial off-the-shelf processor), and the actors communicate and synchronize according to dataflow semantics. This is shown conceptually in Figure 3.3 [98].

### **3.3 Architectures Utilizing Optical Interconnects**

In future CMOS chip designs incorporating hundreds of millions of transistors, the wire interconnect will become a limiting factor, both in terms of area overhead and delay. Optical interconnects offer the potential to relieve this bottleneck. In this section we will summarize some past work in optical interconnects and optically connected architectures, and introduce two new architectures we have developed specifically as a platform on which to map DSP applications described as dataflow graphs.

#### **3.3.1 Optical Interconnect Technology**

In recent years, optics have played an increasing role in multiprocessor systems. Commercial high-performance computers now use fiber ribbons to connect multiple pro-



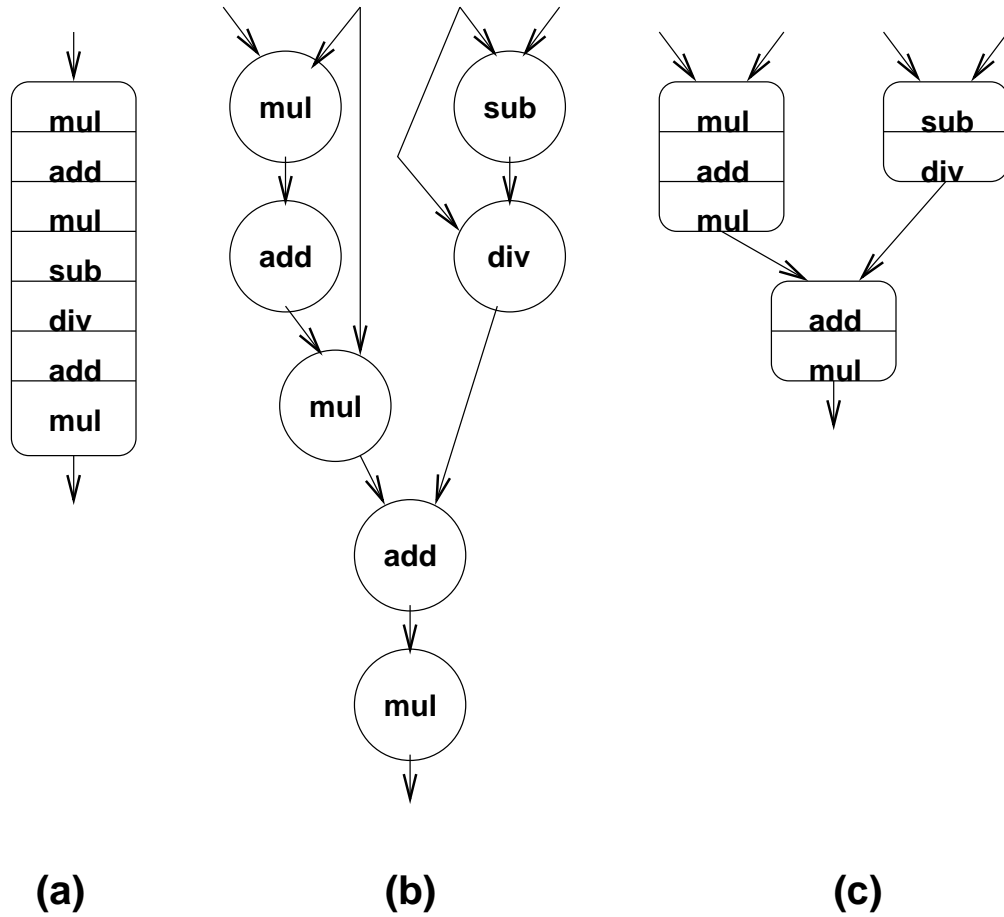


Figure 3.3: Comparison of execution models: (a) von Neumann (control flow) (b) pure dataflow (c) coarse-grain model with dataflow graph and fully ordered grains.

cessing nodes [95]. Other examples include storage area networks using fiberchannel, and optical clock distribution to reduce clock skew rate across a chip [26]. Optical technology has been advancing rapidly, driven in large part by the optical communications equipment market. Various studies have predicted that the energy consumed by data communication will ultimately limit the processing speed in electronic processors [79, 100]. Light signals do not suffer from effects such as electromagnetic interference and capacitive effects, which limit electrical interconnects. While transistor gate delay decreases linearly with decreases in minimum feature size, the wire delay increases as wires become thinner. In addition, the cross-sectional area of metal wires must increase with length to maintain acceptable attenuation. By contrast, an optical channel has a constant transverse area of  $\lambda^2$ , where  $\lambda$  is the wavelength of the light [84]. Thus beyond a certain transmission length, optical interconnections become favorable. This break-even length is estimated to be between  $0.1mm$  and  $1cm$  [63].

There has been theoretical work [37] that has established that arbitrary connection graphs can be realized with an effective interconnection density of  $1/\lambda^2$  using optics. At these densities, heat removal will be the limiting factor [83].

Several studies [36, 65] have addressed the question of what is the best size for a VLSI processor connected by optics. These have concluded that the system should be partitioned into clusters of  $10^4$  to  $10^{12}$  transistors. This allows the design to reach points in the design space that are not achievable without optics. However, there may be significant power and space costs. If size and power are the primary objectives, optical systems become advantageous only with extremely large systems [84].

The main advantage of optics for a multiprocessor system is that it allows highly parallel data links and a large degree of connectivity between processors.

### 3.3.2 Prototype Optically-Connected Systems

Several research groups have demonstrated optically-connected multiprocessor systems (e.g., see [46, 48, 76, 77]). Some of these systems are based on free-space optical interconnects, while others are based on wavelength division multiplexing (WDM). WDM systems typically utilize fiber or waveguide interconnects, and are advantageous for hybrid integration of independent modules. The strength of a free-space optical interconnect scheme is its potential to provide an extremely high density of interconnections, such as will be required for a single-chip system.

An example of a system utilizing free-space optical interconnects is the *FAST-Net* prototype [48]. FAST-Net is a high throughput data switching concept that uses a reflective optical system to globally interconnect a multichip array of processors. The three-dimensional optical system links each chip directly to every other with a dedicated bidirectional parallel data path. The system utilizes smart-pixel arrays (SPA), in which high density silicon electronics are integrated with two-dimensional arrays of high speed Gallium Arsenide micro-laser/detector arrays. An array of SPAs is packaged on a planar substrate and linked to itself through an optical system composed of a lens array and a mirror. This concept provides internal bisection bandwidth [70] on the order of  $10^{12}$  bits per second. Figure 3.4 depicts the SPA and the optical imaging system.

Compiler technology and automated mapping tools for these systems have received relatively less attention than the hardware. Seo and Chatterjee [94] presented a CAD tool for physical placement of modules in SoC utilizing optical interconnects. The tool determined which interconnects should be routed electrically and which should be routed optically. They reported a 50% reduction in worst case interconnect delay over using all metallic interconnects.

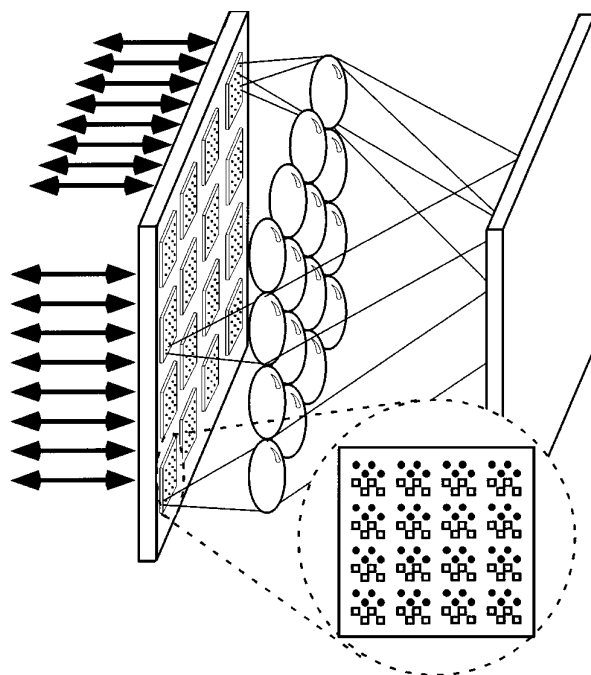


Figure 3.4: FASTNet prototype.

### 3.4 Optically Connected System on Chip

In this section we will examine several fundamental design considerations for systems on chip utilizing optical interconnects. Our general model for a system-on-chip (SoC) is one in which the chip is partitioned into regions that are connected with metallic (local) interconnects, and these local regions are then connected through optical (global) interconnects [11]. As mentioned in the Introduction, the applications we consider can be modeled by dataflow, and consist of task graphs, where the individual tasks must fit fully into a local region. The graph vertices (tasks or nodes) in the acyclic task graphs represent computations while the edges represent the communication of a packet of data from a source task to a sink task.

Three fundamental design considerations for such a system are addressed in this thesis:

- What is the optimum size of a local partition?
- What techniques should we use to map and schedule tasks on these partitions?
- How do we synthesize an optimum global (optical) interconnection network for the system?

These considerations are interrelated, since the size of the local partition will affect the maximum size (granularity) of the tasks, and the scheduling of tasks depends on the interconnection network. This section will focus on the question of optimal partition size. Scheduling is addressed in Chapter 6 and interconnect synthesis is covered in Chapter 7.

### 3.4.1 Global/Local Partitioning

This section presents an information-theoretical model for trade-offs in designing the local partition of a SoC utilizing free-space optics. As mentioned earlier, free-space optical interconnects can provide higher interconnect densities than other types of optical interconnects. These trade-offs are fundamental in nature and will exist in any system utilizing these interconnects.

These systems utilize arrays of vertical cavity surface emitting laser (VCSEL) transmitters and photoreceivers to implement the interconnect. A single interconnect consists of a VCSEL/photoreceiver pair. Light from the VCSEL must be directed to and imaged on the appropriate photoreceiver. This is depicted for the FAST-Net system in Figure 3.5. Different systems use different imaging methods to accomplish this. The high density of interconnections arises from the use of the third dimension (free-space) and the fact that overlapping optical signals do not interfere with each other (i.e., there is no crosstalk in free space).

As the dimensions of the local partition decrease, higher f-number lenses are required to collect the light from the transmitters in a constant focal-length system. (The f-number of a lens is defined as its focal length divided by its diameter). Figure 3.6 depicts the diffraction-limited images of an array of point sources, in a random on/off pattern, on an array of photodetectors. The data for the figure was generated using MATLAB to compute the diffraction pattern for F/1 lenses (left) and F/2 lenses (right). Using an optical system with f-number F and treating the transmitter as a point source operating at wavelength  $\lambda$ , the diffraction-limited image of the source on the detector is given by the expression

$$Ai(\rho) = I_0 \left( \frac{2J_1\left(\frac{\pi\rho}{\lambda F}\right)}{\frac{\pi\rho}{\lambda F}} \right)^2 \quad (3.1)$$

where  $\rho$  is the radius from the center of the image and  $I_0$  is proportional to the source

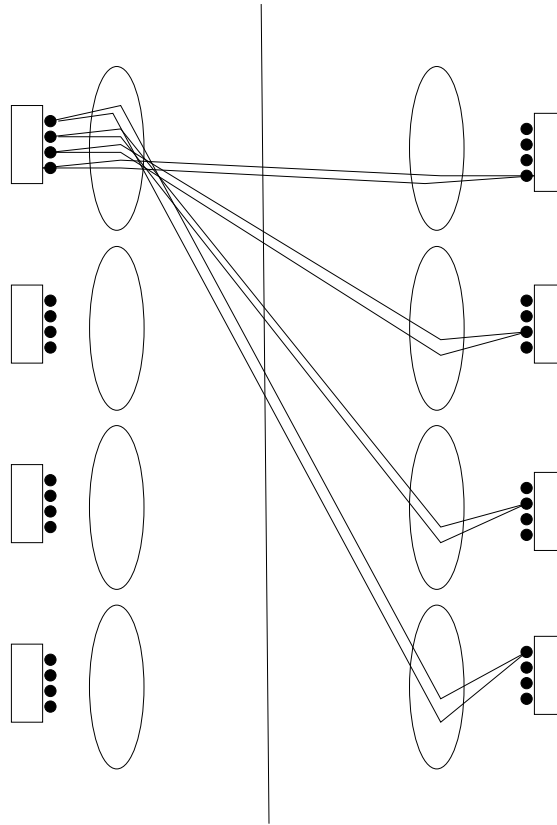


Figure 3.5: Schematic side view of the global optical interconnection shown folded about the mirror plane for the FAST-Net system.

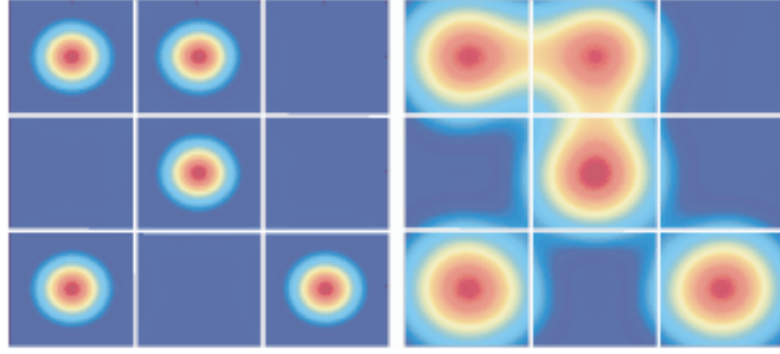


Figure 3.6: An array of point sources imaged using f/1 optics (left) and f/2 optics (right). The left and right pictures are different scales—the partitions on the left are twice the length of the partitions on the right.

intensity. The function  $J_1$  is a first order Bessel function of the first kind.

From this equation, the signal received by the center channel for this pattern can be calculated by spatially integrating over the corresponding photodetector. This calculation will also take into account the inter-pixel interference (IPI). We then vary the pattern randomly to generate the conditional probability distributions for the center channel. If we assume that the IPI is only significant between adjacent channels, we can use the conditional probabilities to assess the mutual information corresponding to a channel between partitions. As partition size decreases, and the associated aperture sizes decrease (increasing the f-number), the optical signal intensity decreases and the IPI increases. Both effects reduce the mutual information. We can then characterize the mutual information as a function of partition size, and therefore, the number of partitions. The mutual information between each source and its corresponding detector is given by

$$I_{mut}(X; Y) = \sum_{i=0,1} p(y|X = i) \log_2 \left[ \frac{p(y|X = i)}{p(y)} \right] dy \quad (3.2)$$

where  $p(y|X = i)$  is the conditional probability that a value  $y$  is received when  $i$  is



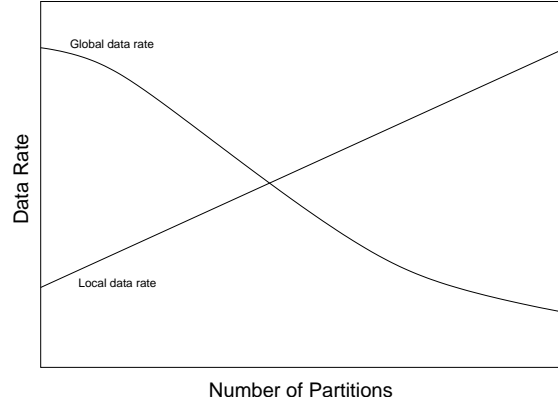


Figure 3.7: Trade-Off between partition size, global data rate, and local data rate.

transmitted and  $p(y)$  is the probability density function (PDF) of  $y$ .

Restoring the mutual information required for the application can be achieved by decreasing the bit rate and integrating over a longer clock cycle in order to increase the signal-to-noise ratio. We define the information capacity, or data rate, as the product of the mutual information and the bit rate. Therefore, it can be generally shown that increasing the number of partitions on a chip will lead to lower global data rate across the chip. At the same time, smaller partitions will reduce the length requirements on local interconnections (intra-partition) performed electrically. Therefore, local interconnect data rates can benefit from reduced partition size. We assume that the data rate is inversely proportional to the RC time constant, which in turn is proportional to the square of the interconnection length. A simple approximation then results in a factor  $\sqrt{N}$  decrease in local interconnect length, therefore, a factor  $N$  increase in the local data rate, where  $N$  is the number of partitions. These opposing effects of partition size suggest a trade-off between the local and global data rates, which is illustrated hypothetically in Figure 3.7, and thus an optimum partitioning of the SoC. This is the crossing point of the two curves in Figure 3.7.

### 3.4.2 Typical Numbers

We next give some estimates of system parameters based on today's components. The optical channel density on the chip will impose a fundamental upper limit on the number of partitions,  $M$ , for the SoC. For a chip with dimensions  $L \times L$ , the number of optical channels  $N$  will be given by  $N \leq L^2/2d^2$  where  $d$  is the VCSEL and detector pitch. For a full crossbar connection,  $N = M(M - 1)$ . For a "typical" VCSEL pitch of 125 microns, this implies that we would be limited to 57 partitions for a one square centimeter chip. The power requirements depend on the architecture, but some insight can be gained by considering examples. Let  $P_0$  represent the power required to drive a VCSEL-detector pair. If every partition is transmitting and receiving data, the total optical power is given by the number of partitions times the number of VCSEL-detector pairs transmitting per partition times  $P_0$ . The upper limit of power consumption corresponds to the case in which all VCSEL-detector pairs are operating. Therefore,  $P \leq L^2/2d^2 P_0$ . The lower limit to the power consumption corresponds to the case where only one pair per partition is transmitting at any instant of time, which implies  $P \geq M P_0$ . If we assume  $P_0 = 10$  mW and 57 partitions, then the total power consumption would be 32W for the one square centimeter chip in the most demanding case and 570mW for the least demanding case.

The one-way data rate between two partitions is given by the data rate per VCSEL-detector pair,  $D_0$ , times the number of pairs:  $D_{\text{partition}} = L^2/2d^2 M(M - 1) D_0$ . For  $D_0 = 2.5$  Gbps,  $D_{\text{partition}} = 4$  Tbps in a two-partition architecture. In the case of a single VCSEL-detector pair per cluster, the partition data rate is equal to the channel data rate at 2.5 Gbps, with an aggregate data rate of 142.5 Gbps for 57 partitions.

### 3.5 Modeling Optically-Interconnected Systems with Synchronization Graphs

A graph-theoretic framework, called the *synchronization graph*, for analyzing arbitrary algorithm-to-architecture mappings is given in [101]. In this section we describe three architectures we developed to take full advantage of the analytical properties of this framework. The synchronization graph applies to any hardware architectural model that includes the following assumptions:

- For each computational task (dataflow node), a reasonably accurate estimate exists for the execution time of a task, and this execution time exhibits little or no variation with input data.
- Once a communication link is reserved for a specific data packet, the link remains reserved exclusively for that packet until transfer of the packet completes.
- The transit time of data packets through the interconnection network, once a communication link has been reserved for the transfer, is deterministic.

If we assume that the time required to perform interprocessor communication is zero, then the synchronization graph work shows that the throughput of a given algorithm-to-architecture mapping can be determined accurately by an efficient graph-theoretic technique [18]. If the interprocessor communication is nonzero, the technique gives an upper bound to the throughput. The tightness of this upper bound depends on the ratio of interprocessor communication time to average task execution time. In optically-connected multiprocessor systems, we can expect that this ratio is small. This is a particularly good assumption if the implementation guarantees that there is never contention among the processors for an optical link. The existence of an accurate, efficient throughput analy-

sis technique opens up the possibility of developing improved algorithm-to-architecture mapping techniques, which we explore in this thesis.

### **3.5.1 SLOT Architecture**

We developed an architecture, which we call *SLOT* (Self-timed Locally Ordered Transaction), on which a broad class of high-throughput, self-timed DSP applications can be analyzed accurately using efficient algorithms based on the synchronization graph framework. *SLOT* enables the development of powerful tools for automatic application mapping (compilation). The interprocessor connectivity requirements of *SLOT* are large, and thus optical interconnect technology appears to be a natural match for *SLOT* systems. In particular, a general-purpose slot architecture requires that each processor have a dedicated communication channel for each processor with which it communicates. Figure 3.8 gives a graphical representation of this architecture.

*SLOT* architectures can be composed of arbitrary, possibly heterogeneous, collections of processing elements, such as DSP processors, FPGA or ASIC subsystems, microprocessors, and microcontrollers. When a processor is embedded within a *SLOT* architecture, one or more communication processors are used to interface the processor to the rest of the multiprocessor system. Each communication processor is assigned a pre-defined ordering of the interprocessor communication operations (send and receive operations to and from other processors) that are required to interface the associated (computation) processor. These local orderings of communication operations, on the communication processors within a *SLOT* system, are repeated over and over again based on the arrival of data (from the associated computation processors, or from other communication processors). A group of communication processors can also be “clustered together” without an associated computation processor. Such clusters of communi-

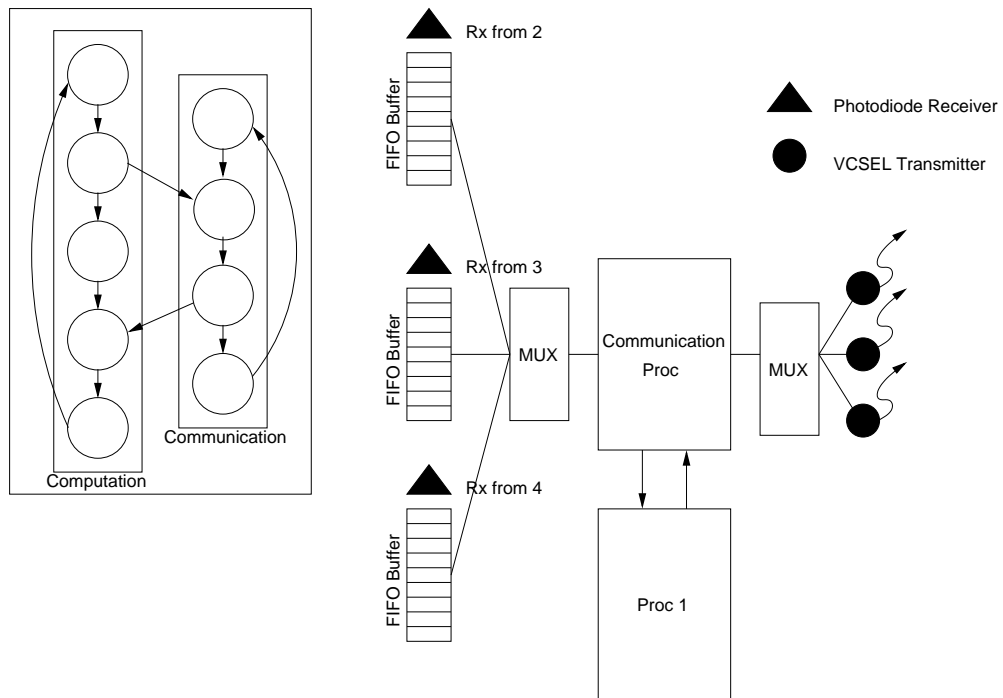


Figure 3.8: Schematic of SLOT architecture.

cation processors serve as routers that provide additional communication paths between remote computation processors. Figure 3.8 depicts one computation and communication processor in a four-processor, fully connected system. A dedicated laser transmitter is required for each other processor with which this processor must send data. Also, a dedicated photodiode receiver and buffer memory is required for each processor from which the processor receives data. With this architecture, there is no contention for communication resources, and the synchronization graph models the system accurately. This architecture is particularly well suited to be implemented in free-space optical systems such as FAST-Net. As mentioned above, one advantage of free space interconnects is the high density of interconnects that can be achieved. If we were to replace the processing element in the FAST-Net prototype with the combination of multiplexers, communication processor, and computation processor from Figure 3.8, SLOT could be implemented using the FAST-Net optical imaging, packaging, and smart pixel array hardware.

### 3.5.2 Dedicated Channel Fiber WDM Architecture

One disadvantage of free-space optical systems is that they are very sensitive to alignment. The alignment of the optical paths described in Section 3.4.1 from each laser transmitter to the correct photodiode receiver may be difficult and not robust under some operating environments (due to vibration, temperature changes, etc.). Fiber-based architectures do not suffer from this problem—the VCSEL-to-fiber and fiber-to-photodiode interface has proven to be very robust in commercial systems. Here we describe a fiber-based implementation of SLOT.

In this implementation, we need to assign a unique wavelength to each communication channel. We define the processor graph  $G_p$  as a directed graph in which the nodes represent the processors in the system and the edges represent connections be-

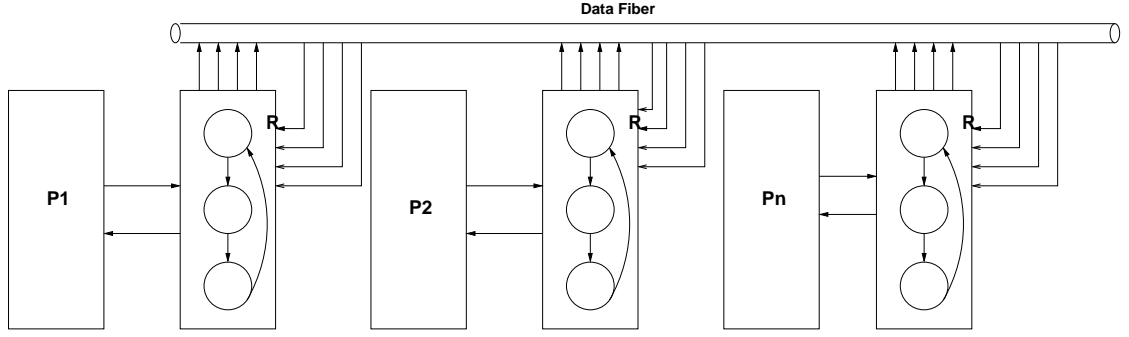


Figure 3.9: Architecture for contention-free fiber-based SLOT.

tween processors—if processor  $i$  transmits data to processor  $j$ , there is an edge  $(i, j)$  in  $G_p$ . This is essentially the WDM equivalent to the free space interconnect since there is a dedicated channel between every pair of processors. Physical constraints will usually place limits on the fan-out and fan-in of the processors. Fan-out of a processor  $p$  is defined as the out-degree of node  $p$  in  $G_p$ , while fan-in is defined as the in-degree of node  $p$  in  $G_p$ . We will define the maximum allowed fan-out as  $f_{\text{out}}$  and the maximum allowed fan-in as  $f_{\text{in}}$ . Figure 3.9 depicts this implementation.

The advantage of this implementation is that there are no central controllers required. This architecture allows a direct implementation of the synchronization graph. A disadvantage is the number of wavelengths required—for a system with  $n$  processors, there are  $n^2$  wavelengths required, or  $n f_{\text{out}}$  wavelengths required if we place a constraint on the fanout.

### 3.5.3 One Wavelength Per Processor

In order to reduce the number of wavelengths required, we can implement a protocol in which each processor is assigned a unique wavelength. In this system, we must ensure that two processors do not send to a given processor at the same time—i.e., there is

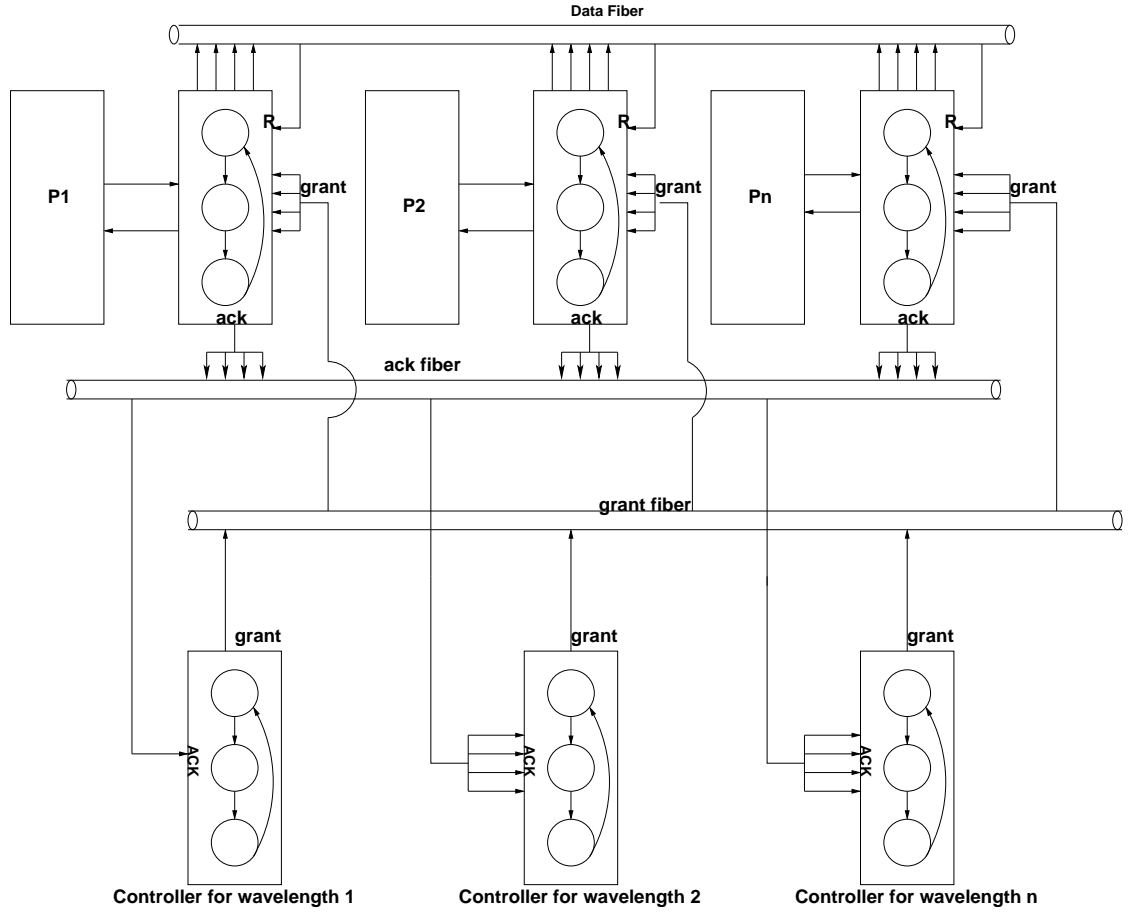


Figure 3.10: Architecture for wavelength ordered transactions.

possible contention at the (single) receiver of every processor. In order to accomplish this, we introduce a controller for every wavelength. This controller grants access to only one processor at a time. Figure 3.10 depicts this implementation.

In this architecture, processor  $m$  receives data on its uniquely assigned wavelength  $\lambda_m$ . In order to grant  $\lambda_m$  to processor  $q$ , the controller for  $\lambda_m$  sends the *number*  $q$  on its grant output (which is at wavelength  $\lambda_m$ ). The controller has an acknowledgment (ACK) receiver for every processor to which it grants access. The communication processors must wait to be granted access to a particular wavelength before transmitting



on that wavelength. The number of grant lines entering a communication processor is equal to  $f_{out}$ , since there must be a grant for each  $\lambda$ , and a processor transmits on a maximum of  $f_{out}$  wavelengths. When a processor  $p$  has completed transmission on a given wavelength, say  $\lambda_r$ , it sends an acknowledgment consisting of the *number*  $r$  on wavelength  $\lambda_p$ . The number of ACK lines entering a wavelength controller  $x$  is equal to the fan-out  $f_{out}$  for processor  $x$ . One advantage of this architecture is that it requires fewer wavelengths— $n$  wavelengths are required as opposed to  $n^2$  or  $n f_{out}$ . One disadvantage is that it is more complicated and  $n$  controllers are required. Also the throughput may be lower since the system is more constrained—we have the same synchronization graph as before with extra edges added for the grants and acknowledgments. We refer to this architecture as *wavelength division multiplexing ordered transactions* (WDMOT).

We will examine the theoretical performance of the three architectures described above in Chapter 5.

## **Chapter 4**

# **Contention Analysis in Shared Bus Systems Utilizing the Period Graph**

### **4.1 Contention in Shared Bus Systems**

In many practical multiprocessor systems, there is contention for one or more shared communication resources. One example of this is a shared bus, in which the processors must first gain access to the bus before they can execute an interprocessor communication (IPC) operation. Figure 4.1 depicts a simple architecture with three processors, a shared memory, and a shared bus. One consequence of this contention is that under self-timed, iterative execution, there is no known method for deriving an analytical expression for the throughput of the system [101], and thus, simulation is required to get a clear picture of application performance. However, simulation is computationally very expensive, and it is highly undesirable to perform simulation inside the innermost optimization loop during synthesis. To avoid such a simulation, an accurate and efficient estimator for throughput is required. In this chapter we will present an efficient estimator for the throughput of these systems when operating in a self-timed, iterative manner. As explained in Chapter 2, in self-timed execution the assignment of tasks to processors

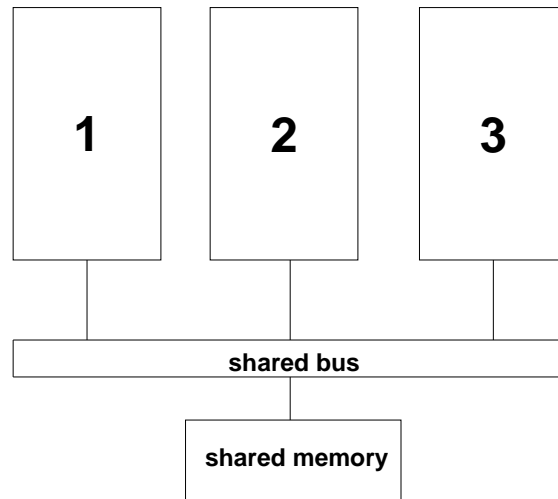


Figure 4.1: Schematic of a three processor shared bus architecture.

and the execution ordering of tasks on each processor are determined at compile-time, and at run-time, processors synchronize with one another only based on inter-processor communication requirements, and do not necessarily synchronize at the end of each loop iteration.

If contention is resolved deterministically, and execution times are constant, then self-timed evolution may lead to an initial transient state, but the execution will eventually become periodic. This holds because the multiprocessor may be modeled as a finite-state system, and thus, aperiodic behavior—which implies the presence of infinitely many states—cannot hold. In DSP systems, although execution times are not always constant or known precisely, they typically adhere closely to their respective estimates with high frequency. Under such conditions, the periodic execution pattern obtained from the estimated execution times provides an estimate of overall system throughput based on the task-level estimates. The estimates for task execution times can be obtained through several methods. The most straightforward is for the programmer to provide them while developing a library of primitive blocks, as is done in the

Ptolemy system [21]. Analytical techniques also exist. Li and Malik [72] have proposed algorithms for estimating the execution time of embedded software tasks in an efficient manner. Due to the largely deterministic nature of DSP applications, such system-level performance analysis, and optimization based on task-level estimates is common practice in the DSP design community [25, 35, 50, 68].

For self-timed systems, when we apply execution time estimates to estimate overall throughput, it is necessary to simulate (using the execution time estimates) past the transient state until a periodic execution pattern (steady state) emerges. Unfortunately, the duration of the transient may be exponential in the size of the application specification [101], and this makes simulation-intensive, iterative synthesis highly unattractive.

We introduced the novel *period graph* model [9] in order to greatly reduce the rate at which simulation must be carried out during iterative synthesis. Given an assignment  $\nu$  of task execution times, and a self-timed schedule, the associated period graph is constructed from the periodic, steady-state pattern of the resulting simulation. The maximum cycle mean (MCM) of the period graph (with certain adjustments) is then used as a computationally-efficient means of estimating the iteration period (the reciprocal of the throughput) as changes are explored within a neighborhood of  $\nu$ . In this context, the MCM is the maximum over all directed cycles of the sum of the task execution times divided by the sum of the edge delays. The MCM can be computed in low polynomial time [66].

## 4.2 Constructing the Period Graph

The first step in the construction of the period graph is the identification of the period from the simulator output. This can be performed by tracing backward through the

simulation and searching for the latest intermediate time instant  $t_a$  at which the *system state*  $S(t_a)$  equals the state  $S(t_f)$  obtained at the end of the simulation (here,  $t_f$  denotes the simulation time limit). If no match is found, then the end of the first period exceeds  $t_f$ , and thus, the simulation needs to be extended beyond  $t_f$ . Otherwise, the region (often depicted as a Gantt chart) that spans the interval  $[t_a, t_f]$  constitutes a (minimal) period of the simulated steady state.

Here, the system state  $S(t)$  contains the execution state of each processor, which is either “idle” or representable by an ordered pair  $(A, \tau)$ , where  $A$  is the task being executed at time  $t$ , and  $\tau$  denotes the time remaining until the current invocation of  $A$  is completed. The state  $S(t)$  also contains the current buffer sizes of all IPC buffers, as well as any information (e.g., request queue status) that is used by the protocol for resolution of communication contention. Our approach to efficiently determining the period is as follows:

- Perform a simulation of the schedule for some time  $T_{sim}$ . Define a constant  $C$ , which is an initial estimate for the number of complete cycles (invocations) of the graph that must be simulated in order to find a period. this constant represents the length of the initial transient, before the output becomes periodic. If this initial estimate is too low, it will be increased during the algorithm. Let  $N$  be the number of processors, and let  $n_j$  be the number of tasks scheduled on processor  $j$ , where  $j \in [1, N]$ . Tasks include IPC tasks as well as computational tasks. Label these tasks  $V_{1_j}, V_{2_j}, \dots, V_{n_j}$ . We consider the case where the system executes these tasks infinitely. The *invocation number* of a task is defined as the number of times a given task has executed, and is denoted with a superscript. For example,  $V_{a(j)}^{b(j)}$  denotes the  $b_{th}$  invocation of task  $a$  on processor  $j$ . Define a simulation array for each processor  $Sim_j[i]$  where  $i \in [1, M_j]$  and  $M_j$  is the number of tasks on

processor  $j$  that were output by the simulator. The elements of the simulation array are the tasks, and are ordered by reverse start time, so that  $\text{Start}(\text{Sim}_j[i]) > \text{Start}(\text{Sim}_j[i + 1])$ .

- Create two *idle vectors* of length  $n_j$  for each processor spanning one invocation. Label the first idle vector  $\text{Idle1}_j^1[k]$  where  $k \in [1, n_j]$ . Label the second idle vector  $\text{Idle2}_j^1[k]$ .
- Examine the *IPC buffer vector* at some fixed point of each idle vector. The IPC buffer vector consists of the numbers of tokens queued on all the IPC edges of the graph enumerated in some order. The IPC buffer vector must be output by the simulator at least once every graph iteration. For example, the simulator could output an IPC buffer vector for each processor every time the processor executes the first task scheduled on it. In this way, each idle vector would be associated with one IPC buffer vector. Label these vectors  $\text{IPCBuf1}_j[q]$  and  $\text{IPCBuf2}_j[q]$  where  $q \in [1, E]$  and  $E$  is the number of edges in the IPC graph. The IPC buffer vector represents the state of the communication buffers in the system. Let  $\text{Tokens}(e, t)$  be the number of data tokens on edge  $e$  at time  $t$ . Let  $\text{TaskNum}_j(t)$  be the number of the node that is executing on processor  $j$  at time  $t$ . Pseudo-code from [10] for constructing the period graph is shown in Figure 4.2.

Our experience suggests that in practice, most graphs have periods spanning only a few invocations, so the above procedure for finding the period is efficient. For a system with a period that spans  $N$  invocations and with at most  $L$  tasks per processor, this method requires  $LN(N + 1)$  comparisons.

Figure 4.3(a) illustrates an application graph, Figure 4.3(b) illustrates a self-timed schedule, Figure 4.3(c) shows the periodic steady state that results from the schedule

**Algorithm 4.1:** CALCULATE PERIOD( $C, T_{sim}$ )

```

 $min_c \leftarrow 0$ 
while  $min_c < C$ 
  do  $\left\{ \begin{array}{l} \text{INCREMENT}(T_{sim}) \\ \text{SIMULATE}(T_{sim}) \\ min_c \leftarrow \min \left( \lfloor \frac{M_j}{n_j} \rfloor \right) \end{array} \right.$ 
for  $t \leftarrow 0$  to  $T_{sim}$ 
  do  $\left\{ \begin{array}{l} \text{for } j \leftarrow 1 \text{ to } N \\ \text{do } \left\{ \begin{array}{l} a_j = TaskNum_j(t) \\ invocation_j[a] \leftarrow invocation_j[a] + 1 \\ b_j = invocation_j[a] \\ \text{if } TaskNum_j(t) > TaskNum_j(t-1) \\ \text{then } \{ Sim1_j[i] = V^b(j)_a(j) \} \end{array} \right. \end{array} \right.$ 
 $span = 0$ 
repeat
   $\left\{ \begin{array}{l} span \leftarrow span + 1 \\ \text{for } k \leftarrow 1 \text{ to } span * n_1 \\ \text{do } \left\{ \begin{array}{l} \text{for } j \leftarrow 1 \text{ to } N \\ \text{do } \left\{ \begin{array}{l} \text{if } span * n_j > M_j \\ \text{then } \{ \text{comment: error: increase C and start over} \} \\ Idle_j^1[k] = Finish(Sim_j[k]) - Start(Sim_j[k+1]) \\ Idle_j^2[k] = Finish(Sim_j[span * n_j + k]) - Start(Sim_j[span * n_j + 1 + k]) \end{array} \right. \\ \text{for } q \leftarrow 1 \text{ to } E \\ \text{do } \left\{ \begin{array}{l} IPCBuf1[q] = Tokens(q, Start(Sim_1[1])) \\ IPCBuf2[q] = Tokens(q, Start(Sim_1[span * n_1 + 1])) \end{array} \right. \end{array} \right. \end{array} \right.$ 
until  $\prod_j (Idle_j^1 \equiv Idle_j^2) = 1$  and  $(IPCBuf1 = IPCBuf2)$ 

```

Figure 4.2: Pseudocode for constructing the period graph.

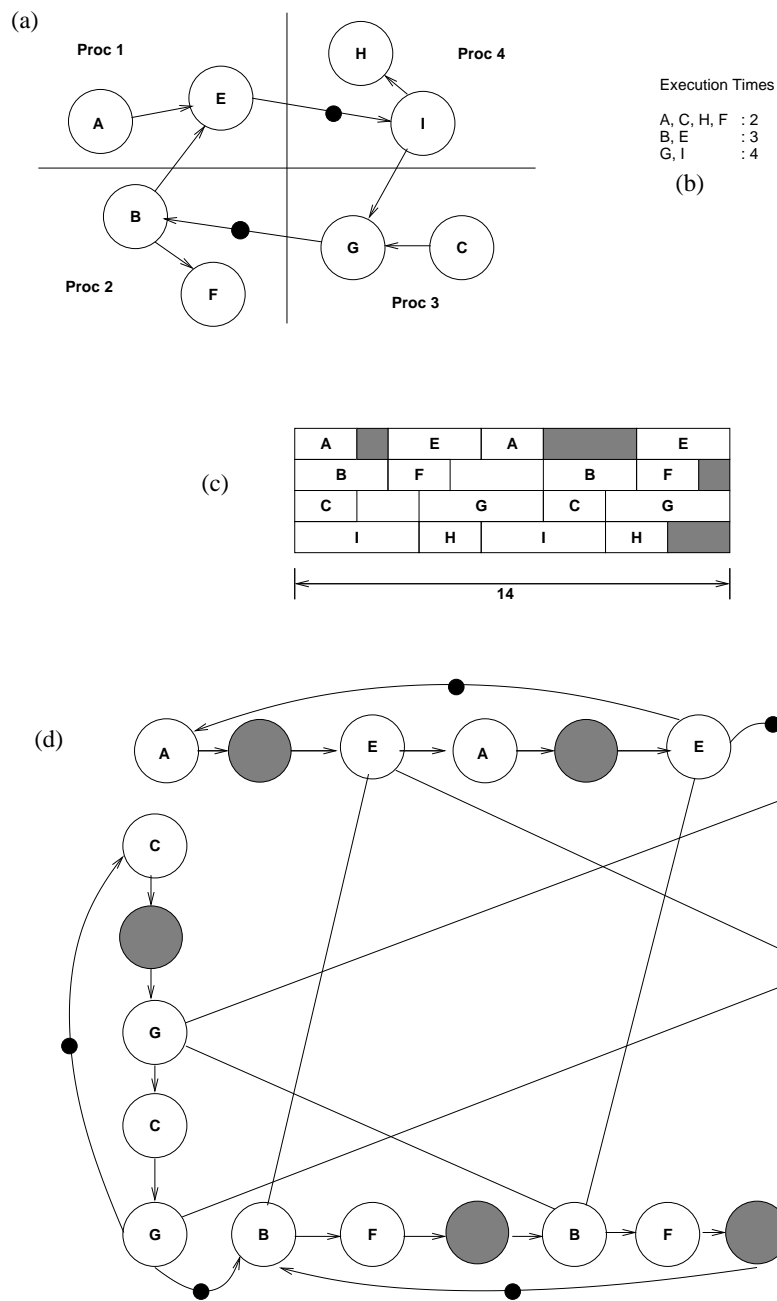


Figure 4.3: An illustration of the period graph construct.



and execution time estimates, and Figure 4.3(d) depicts the resulting period graph. The shaded nodes in Figure 4.3(d) correspond to idle time ranges in the period, and solid black circles on edges represent delays, which model inter-iteration dependencies. Note that the steady state period may span multiple graph iterations (two in this example), and in the period graph, this translates to multiple instances of each application graph task.

For clarity in this illustration, we have assumed negligible latency associated with IPC. As described below, non-negligible IPC costs can easily be accommodated in the period graph model by introducing *send* and *receive* tasks at appropriate points.

As illustrated in Figure 4.3, the period graph consists of all the tasks comprising the period that was detected, with the idle time ranges between tasks (including those that are caused by communication contention) also treated as nodes in the graph. The nodes are connected by edges in the order that they appear in the period. An edge is placed from the last node in the period for each processor to the first node in the period. This edge is given a delay value of one (to model the associated transition between period iterations), while all of the other intraprocessor edges have delay values of zero. This is done for all the processors in the system. Our model utilizes *send* and *receive* nodes for IPC as described above. For each IPC point, a send node is placed on the processor that is sending data, and a corresponding receive node is placed on the processor that will receive the data. The period graph is completed by adding an edge from each send node to its corresponding receive node.

### 4.3 Fidelity of the Estimator

As mentioned above, the period graph can be used to estimate the system throughput of a given self-timed schedule as the task execution times are varied. In order to make a concrete example, we will examine *voltage scaling*. Some processors have the ability to alter their execution voltage while in operation. This allows the processor to operate at an optimal energy/efficiency point. When the voltage on a processor is varied, the execution time of a computational task varies according to

$$\text{delay} = k \frac{V_{dd}}{(V_{dd} - V_t)^2}, \quad (4.1)$$

where  $V_{dd}$  is the supply voltage,  $V_t$  is the threshold voltage, and  $k$  is a constant [24]. We use a value of 0.8volts for the threshold voltage. The execution time  $pe_i$  of each of these states in the original (non-scaled) period graph is referenced to a voltage  $V_{ref}$ . The change in execution time of each computational node is found by taking the derivative:

$$\Delta pe_i = pe_i \left( \frac{V_{sc}}{V_{ref}} \left[ \frac{1 - \frac{V_t}{V_{sc}}}{1 - \frac{V_t}{V_{ref}}} \right]^2 - 1 \right)$$

where  $V_{sc}$  is the new voltage. It is not obvious, however, how one should adjust the idle times in the period graph. We separate the idle nodes into two sets: *contention idles* and *data idles*. When a node has the necessary data to execute (the necessary data has already been produced), but is idle waiting for access to the bus, the associated idle node is classified as a contention idle. When a node is idle waiting for its predecessors' data, the associated idle node is classified as a data idle. By experimenting with a large number of application graphs, we found that we could capture the effects of contention and obtain the best fidelity by zeroing out the data idles and leaving the contention idles constant as the computation idles are scaled. Using these rules, the fidelity is calculated as follows:

- Given an application graph, construct a valid schedule. We used the dynamic level scheduling algorithm given by Sih and Lee [97]. Next, construct the period graph as discussed earlier. Generate  $N$  voltage vectors (assignments of voltages to the processors in the target architecture). For each voltage vector, perform a simulation to determine the throughput, with the execution times of the tasks on each processor given by 4.1 according to the voltage on the processor. Also, obtain an estimate for the throughput by calculating the MCM of the voltage-scaled period graph, in which the execution times of the computation nodes are given by 4.1, and the execution times of the idle nodes are as explained above.
- Calculate the fidelity according to:

$$\text{Fidelity} = \frac{2}{N(N-1)} \sum_{i=1}^{N-1} \sum_{j=i+1}^N f_{ij} \quad (4.2)$$

where

$$f_{ij} = \begin{cases} 1 & \text{if } \text{sign}(S_i - S_j) = \text{sign}(M_i - M_j) \\ 0 & \text{otherwise} \end{cases}$$

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

The  $S_i$  denote the simulated throughput values; and the  $M_i$  are the corresponding estimates from the period graph.

Figure 4.4 plots Fidelity for a six-processor system in which the voltage on the individual processors can vary between plus or minus five percent. The x-axis represents the sum of the absolute values of the voltage changes over all processors. Each point on the graph is a fidelity calculation for  $N = 100$  voltage vectors. A value of one

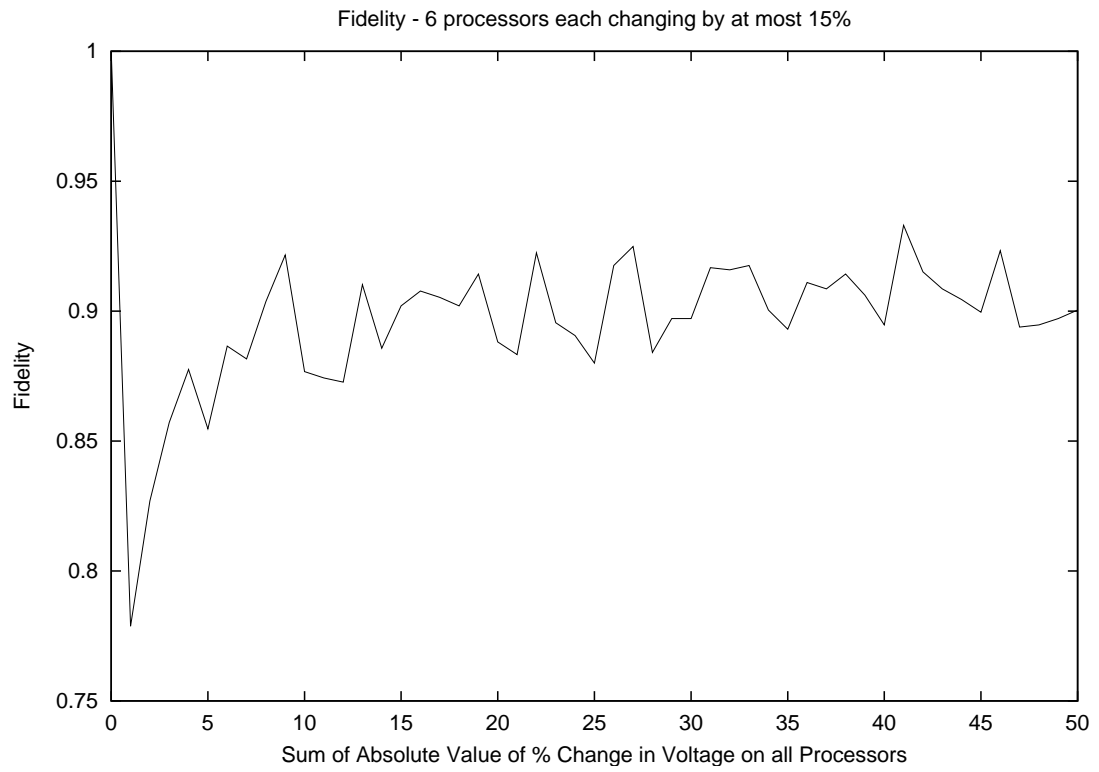


Figure 4.4: Plot of fidelity (equation 4.2) for a six processor system vs. magnitude of voltage change on all processors.

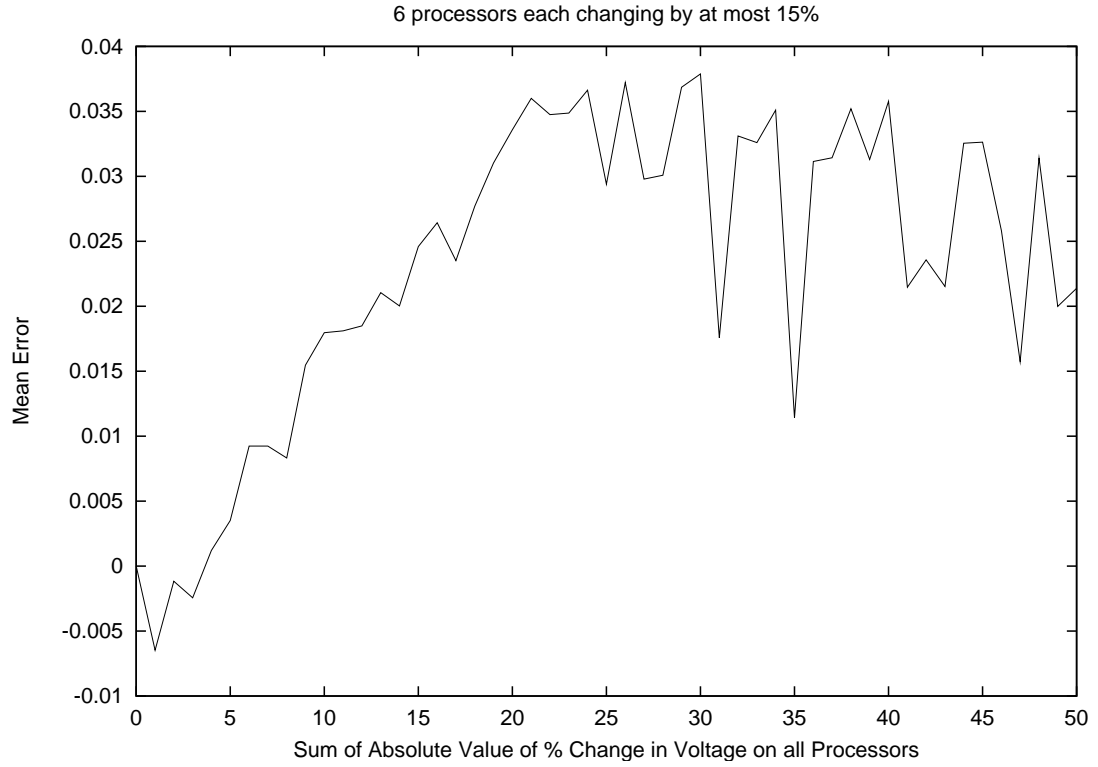


Figure 4.5: Plot of average error (equation 4.3) of fidelity estimate for a six processor system vs. voltage change on processors.

is a “perfect” fidelity. It can be seen that in the range shown, the fidelity is always greater than 0.77. It is also important that the estimator have a small error at each point.

Figure 4.5 plots

$$\sum_{i=1}^N [(S_i - M_i)/S_i] \quad (4.3)$$

for a six processor system. It can be seen that the error increases as the voltage vector moves away from the reference point, and that the estimate is slightly biased. For the range shown in the graphs, where each processor voltage is changed by a maximum of fifteen percent, the error is less than four percent.

## 4.4 Using the Period Graph in a Joint Power/Performance

### Algorithm

An effective way to reduce power consumption of a processor core in CMOS technology is to lower the supply voltage level, which exploits the quadratic dependence of power on voltage [24]. Reducing the supply voltage also has the effect of decreasing the clock speed and increasing circuit delay. The circuit delay can be modeled by 4.1. The power consumption is given by

$$P = \alpha C_L V_{dd}^2 f \quad (4.4)$$

where  $f$  is the clock frequency,  $C_L$  is the load capacitance, and  $\alpha$  is the switching activity [24]. To accommodate the possibility of putting processors in states of lower switching activity during idle periods, our model includes a parameter  $\alpha_{\text{idle}}$  for the idle states, and a parameter  $\alpha_{\text{non-idle}}$  for the computational tasks, where  $\alpha_{\text{idle}} \leq \alpha_{\text{non-idle}}$ . A more detailed power analysis could assign a different  $\alpha$  for each computational task if that data were available. A different power optimization technique, which can be used in conjunction with the voltage scaling technique presented here, utilizes a nearly complete processor shutdown during the idle periods [52, 103]. In our model, this would correspond to  $\alpha_{\text{idle}} = 0$ . Our model for the power is the average energy consumption per graph iteration period. This corresponds in a typical DSP system to the average energy required to process one sample. Here, the energy of each node equals its power times its execution time.

In a system consisting of multiple processors, one has the ability to choose, within a certain range, the (fixed) operating voltage on each processor. This opens up an additional degree of freedom that can be exploited to minimize the system power consumption. By choosing a lower voltage of a processor that is executing tasks that are not

on the critical path, the throughput can remain unchanged while the overall power consumption is reduced. In general, a combination of raising voltages on some processors while lowering others can yield the most attractive power/performance solution.

When applying voltage scaling to a multiprocessor system, the valid solution space is typically much too large to search by brute-force methods. In addition, since there is no general analytical formula for calculating the throughput of these systems in the presence of communication resource contention, each candidate solution must either be simulated or estimated using some heuristic.

## 4.5 Genetic Algorithm Formulation

To demonstrate the general utility of the period graph based performance estimation approach, we incorporated it into two significantly different probabilistic search techniques to derive two different algorithms for systematic voltage scaling [7]. The first algorithm presented utilizes the framework of genetic algorithms (GAs) [43]. We will discuss GAs more in Chapter 8. The specific GA explored here consists of an inner GA nested within an outer GA. The inner GA performs a local search around a point from the population of the outer GA, using the MCM of the period graph in its objective function as an estimate for the throughput. A period constraint  $T_{\text{constraint}}$  is given as an input to the optimization problem, where the period is the reciprocal of the throughput. The objective function calculates the power consumption associated with each solution by calculating the total energy per period, as discussed earlier. If the period associated with a solution violates the period constraint  $T_{\text{solution}} > T_{\text{constraint}}$ , the power consumption is multiplied by a large penalty factor  $e^{100(T_{\text{solution}} - T_{\text{constraint}})}$ . The GA attempts to minimize this objective function.

In the outer loop, a population of  $N_{\text{outer}}$  voltage vectors is generated. A simulation is run and a period graph constructed for each of these outer loop voltage vectors. For each of the outer loop voltage vectors, a new inner loop population is generated such that  $|V_{\text{outer}_i} - V_{\text{inner}_i}| < \epsilon$  for  $i \in N_{\text{proc}}$  where  $N_{\text{proc}}$  is the number of processors,  $V_{\text{outer}_i}$  is the voltage on processor  $i$  in the outer population,  $V_{\text{inner}_i}$  is the voltage on processor  $i$  in the inner population, and  $\epsilon$  is a user-defined threshold. The inner population size is  $N_{\text{inner}}$ . The inner GA then performs a local search using this population for a number of generations  $\text{Generations}_{\text{inner}}$  in an attempt to find a locally optimal voltage vector. The inner GA uses the MCM of the period graph in its objective function. After an invocation of the inner GA is finished, one simulation is performed using the resulting voltage vector, and the actual throughput for this point is used to compute its fitness. The outer loop voltage vector is then replaced with this locally-optimized voltage vector for use in the next outer loop generation. The outer loop is run for a number of generations  $\text{Generations}_{\text{outer}}$ .

## 4.6 Simulated Annealing Algorithm

Simulated annealing is another well-known method for searching large design spaces. Using a standard simulated annealing package [23], we have implemented an alternative version of period-graph-based voltage scaling optimization. The objective function here is the same as for the genetic algorithm. The system is first simulated with an initial voltage vector  $V_j = \text{LSV}_j$ , and the period graph is built. In order to insure that the period graph will be a good enough estimator, a *re-simulation threshold*  $T$  is maintained. The difference between the current input  $\text{CV}_j$  to the objective function, and the voltage vector  $\text{LSV}_j$  corresponding to the simulation used to compute the current period, is



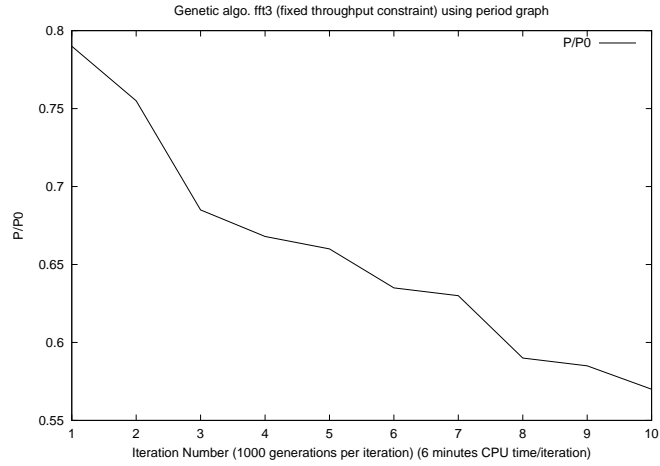
calculated. If

$$\frac{1}{N} \sum_{i=1}^N \left| \frac{V_i - \text{LSV}_i}{\text{LSV}_i} \right| > T,$$

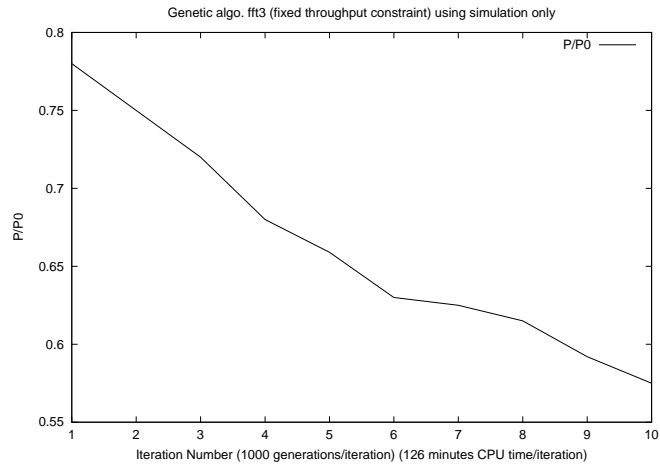
the graph is re-simulated using  $\text{CV}_j$ . The period graph is rebuilt, and  $\text{CV}_j \rightarrow \text{LSV}_j$ . For  $T = 0$ , the graph will be re-simulated every time, and the period graph will offer no speedup to the optimization. The larger the value of  $T$ , the less often the graph will be re-simulated, and the faster the optimization algorithm will perform. However, when  $T$  is too large, the fidelity of the period graph estimate will be unacceptably low and the quality of the final result will suffer. Based on our experiments with a number of graphs, the optimal value of  $T$  is application-dependent, but a value of  $T = 0.1$  generally gives good results.

## 4.7 Results of Voltage Scaling using Period Graph

Figure 4.6(a) shows an example of the reduction in power resulting from the genetic optimization algorithm on the FFT2 application graph. The parameters of the GA were  $N_{\text{outer}} = N_{\text{inner}} = 50$ ,  $\text{Generations}_{\text{outer}} = 10$ , and  $\text{Generations}_{\text{inner}} = 20$ . The local search voltages were constrained to be within five percent of the corresponding outer loop voltages. The period constraint was calculated by simulating the system with all six processors operating at voltage  $V_{\text{ref}}$ . For this example, the system power consumption was reduced by 43%, while maintaining the original throughput. To evaluate the advantage of the period graph approach over using brute-force simulation, a second nested GA was implemented. This algorithm was identical to the algorithm discussed above, except that the inner loop did not use the period graph estimate for the throughput. Instead, each voltage vector was evaluated by simulation. This algorithm consumed 21 times more CPU time, and produced similar results, as shown in Figure 4.6(b).



(a) Using period graph. Each iteration requires 6 minutes CPU time.



(b) Using simulation only. Each iteration requires 126 minutes CPU time.

Figure 4.6: Plot of (optimized power)/(initial power) vs. genetic algorithm iteration using the period graph estimator (a) and simulation only (b). Using simulation only, the iterations require 21 times more CPU time.

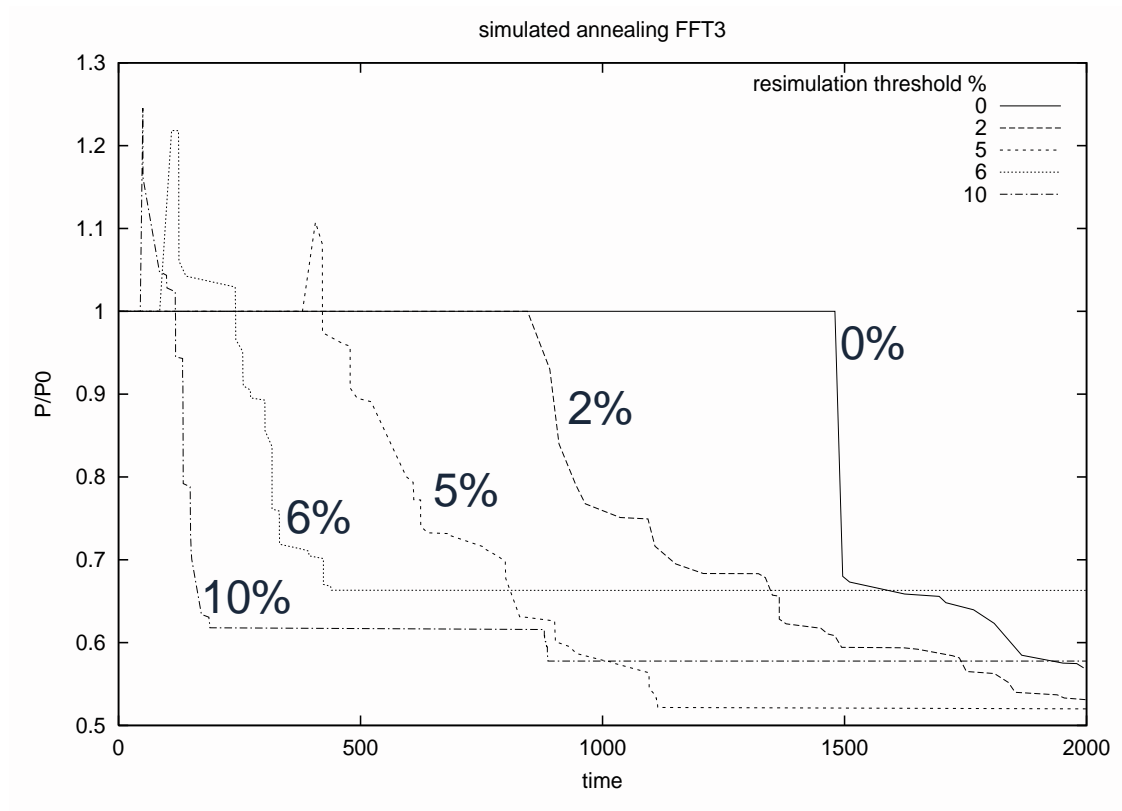


Figure 4.7: Plot of (optimized power)/(initial power) vs. time for the simulated annealing algorithm combined with period graph on the FFT3 application.

Figure 4.7 summarizes the power reduction results for the simulated annealing algorithm applied to a fast Fourier transform (FFT3) application graph, for different values of the re-simulation threshold  $T$ . It can be seen that as  $T$  is increased, the algorithm progresses more quickly. The simulated annealing algorithm begins with a ‘melting’ routine, where the temperature is increased until a phase change is detected. The initial flat part of the curves corresponds to the time spent in the melting routine. We have found that for values of  $T$  above 20%, the period graph is not a good enough estimator and the algorithm does not converge.

Table 4.1 summarizes the power reduction for the simulated annealing algorithm for

application	re-simulation threshold				
	0	2%	5%	10%	25%
-					
FFT1(28)	0.96	0.95	0.65	0.60	1
FFT2(28)	0.97	0.90	0.71	0.97	1
FFT3(28)	1	0.77	0.59	0.59	1
mus(20)	0.89	0.71	0.67	0.82	1
meas(12)	0.77	0.73	0.81	0.82	1
qmf(14)	0.84	0.65	0.67	0.73	1
rand1(30)	0.91	0.77	0.53	0.65	1
rand2(100)	1	0.85	0.77	0.73	1
rand3(200)	1	1	1	0.94	1

Table 4.1: Ratio of optimized power to initial power for a fixed computation time using period graph and simulated annealing.

several additional applications using different values of the re-simulation threshold. At the start of the optimization, all processor voltages were set at 5 volts. The throughput at this point was used as the throughput constraint. In Table 4.1, the first three rows correspond to three different FFT implementations: *mus* refers to a music synthesis algorithm, *qmf* refers to a quadrature mirror filter bank, *meas* is a measurement application.

We implemented a random application graph generator based on Sih’s algorithm [96]. The last three rows of Table 4.1 correspond to three random graphs generated with this algorithm. The numbers in parentheses give the numbers of nodes in these applications. The optimization was performed for a *fixed time* of 30 minutes in each case. The optimum re-simulation threshold was between 2% and 10% in all cases. For  $T = 0.25$ , the period graph is not a good estimator and none of the results returned during the optimization algorithm satisfied the throughput constraint. For the largest graph, the fixed simulation time was not long enough to make much improvement, but the best result occurred for  $T = 0.1$ , where the simulations are less frequent.

Table 4.2 summarizes the power reduction for the genetic algorithm with and without using the period graph, with a fixed compile time (run time) of one hour. It can be seen that, under the condition of fixed compile time, we achieve better results (lower power) when utilizing the period graph. Also, comparing Table 4.1 with Table 4.2, we see that the longer compile time given to the GA produced better results. We will explore the issue of search efficiency under fixed optimization times in a systematic manner in Chapter 8.

<b>application</b>	<b>using period graph</b>	<b>no period graph</b>
fft1	0.54	0.74
fft2	0.69	0.86
fft3	0.57	0.78
mus	0.68	0.90
meas	0.70	0.82
qmf	0.64	0.84
rand1(30)	0.55	0.78
rand2(100)	0.70	1
rand3(200)	0.87	1

Table 4.2: Ratio of (optimized power)/(initial power) for genetic algorithm with fixed run time.

## 4.8 Summary of Period Graph Work

We have developed a *period graph* model that can be used as a computationally efficient estimator for the throughput in multiprocessor systems in which communication contention renders exact analysis too time-consuming. This model is especially useful in interactive synthesis techniques, such as those based on probabilistic search. We demonstrated effective voltage scaling techniques based on incorporating the period graph into genetic algorithm and simulated annealing formulations.

## Chapter 5

# Contention Analysis in Optically Connected Systems

We introduced the IPC graph model in Section 2.4 and showed that for systems without contention, the maximum cycle mean (MCM) of the IPC graph can be used as an efficient estimator for the system throughput. We showed in Chapter 4 that for a shared-bus system the analysis is complicated significantly by contention for the bus among the processors. Shared bus systems are appealing due to their simplicity and low cost. This is the primary driver for many embedded systems applications. In Section 5.1 we will discuss Sriram’s *ordered transactions strategy* [102], and show that by incorporating an additional hardware controller to a shared bus system, it is possible to remove the contention that results in the difficult analysis, and to more fully optimize communication patterns. With the hardware controller the processors still share a communication channel, namely the bus, but the contention is resolved by the controller. For systems that require the performance, the cost of the additional hardware may be justified.

For systems with significant interprocessor communication activity and high performance requirements, it may be the case that an electronic interconnect between the processors is not appropriate. In Section 3.5.2 we introduced two system architectures

based on fiber interconnects. In the first architecture, there is a dedicated communication channel between every pair of processors so there is no sharing of communication resources. In the second architecture, processors share their input ports—the amount of sharing is reduced as compared to an electronic bus because different wavelengths can be transmitted simultaneously in the fiber. In Section 5.2 we will modify Sriram’s model to work with this architecture.

## **5.1 Ordered Transactions**

### **5.1.1 Ordered Transactions Concept**

The ordered transactions strategy for multiprocessor shared bus systems consists of two parts:

1. Determine at compile time the order in which processor communications occur.
2. Enforce that order at run time with a hardware controller.

As in the self-timed approach, a static schedule is first computed using execution time estimates for the actors, but only the actor ordering on each processor is retained—the actor start times are discarded. The hardware controller grants access to the processors in the predetermined order. When a processor is granted access to the bus, it performs its read or write operation and releases the bus back to the hardware controller. Since the hardware controller enforces the communication order, there is no contention for the bus, and no bus arbitration is necessary at the individual processors. The transaction order preserves the data precedences in the algorithm, and therefore for a shared memory system no semaphore synchronization is necessary. Also, send and receive operations



always access the shared bus for one memory cycle—there is no polling required. This reduces the number of shared memory accesses by at least a factor of two.

### 5.1.2 Synchronization Constraints

We introduced the interprocessor communication graph (IPC graph)  $G_{\text{IPC}}$  and the synchronization graph  $G_s$  in Section 2.4. Initially,  $G_s$  is identical to  $G_{\text{IPC}}$ . However, various transformations can be applied to  $G_s$  in order to make the overall synchronization structure more efficient [101]. After all transformations on  $G_s$  are complete,  $G_s$  and  $G_{\text{IPC}}$  can be used to map the given parallel schedule into an implementation on the target architecture. The IPC edges in  $G_{\text{IPC}}$  represent buffer activity, and are implemented as buffers in shared memory, whereas the synchronization edges of  $G_s$  represent synchronization constraints, and are implemented by updating and testing flags in shared memory. If there is an IPC edge as well as a synchronization edge between the same pair of tasks, then a synchronization protocol is executed before the buffer corresponding to the IPC edge is accessed to ensure sender-receiver synchronization. On the other hand, if there is an IPC edge between two tasks in the  $G_{\text{IPC}}$ , but there is no synchronization edge between the two, then no synchronization needs to be done before accessing the shared buffer. If there is a synchronization edge between two tasks but no IPC edge, then no shared buffer is allocated between the two tasks; only the corresponding synchronization protocol is invoked.

Any transformation we perform on the synchronization graph must respect the synchronization constraints implied by  $G_{\text{IPC}}$ . If we ensure this, then we only need to implement the synchronization edges of the optimized synchronization graph (in conjunction with the IPC edges of  $G_{\text{IPC}}$ ). If  $G_1 = (V, E)$  and  $G_2 = (V, E)$  are synchronization graphs with the same vertex-set and the same set of intraprocessor edges (edges that

are not synchronization edges), we say that  $G_1$  *preserves*  $G_2$  if for all  $e \in E_2$  such that  $e \notin E_1$ , we have

$$\rho_{G_1}(\text{src}(e), \text{snk}(e)) \leq \text{delay}(e),$$

where  $\rho_G(x, y) \equiv \infty$  if there is no path from  $x$  to  $y$  in the synchronization graph  $G$ , and if there is a path from  $x$  to  $y$ , then  $\rho_G(x, y)$  is the minimum over all paths  $p$  directed from  $x$  to  $y$  of the sum of the edge delays on  $p$ . Thus,  $G_1$  preserves  $G_2$  if for any new edge in  $G_2$  (i.e., for any edge not in  $G_1$ ), there is a path in  $G_1$  directed from the source of the edge to the sink that has a cumulative delay that is less than or equal to the delay of the edge. The following theorem (developed in [101]) is fundamental to synchronization graph analysis.

**Theorem 1** The synchronization constraints in a synchronization graph

$G_s^1 = (V, E_{\text{int}} \cup E_s^1)$  imply the synchronization constraints of the synchronization graph  $G_s^2 = (V, E_{\text{int}} \cup E_s^2)$  if the following condition holds:  $\forall \epsilon \text{ s.t. } \epsilon \in E_s^2, \epsilon \notin E_s^1, \rho_{G_s^1}(\text{src}(\epsilon), \text{snk}(\epsilon)) \leq \text{delay}(\epsilon)$ ; that is, if for each edge  $\epsilon$  that is present in  $G_s^2$  but not in  $G_s^1$ , there is a minimum delay path from  $\text{src}(\epsilon)$  to  $\text{snk}(\epsilon)$  in  $G_s^1$  that has total delay of at most  $\text{delay}(\epsilon)$ .

Theorem 1 is the basis for a variety of useful synchronization graph transformations. One such transformation is the detection and removal of *redundant* synchronization edges, which are synchronization edges whose respective synchronization functions are subsumed by other synchronization edges, and thus need not be implemented explicitly. Another transformation, called *resynchronization*, involves inserting synchronization edges in a way that the number of original synchronization edges that become redundant exceeds the number of new edges added.

### 5.1.3 Ordered Transactions Graph

Sriram's *ordered transaction graph* [102] is a useful data structure for analyzing ordered transaction implementations. Given an ordering  $\{o_1, o_2, \dots, o_p\}$  for the communication actors in an IPC graph  $G_{IPC} = (V_{IPC}, E_{IPC})$ , the corresponding ordered transaction graph  $\Gamma(G_{IPC}, O)$  is defined as the directed graph  $G_{OT} = (V_{OT}, E_{OT})$ , where

$$V_{OT} = V_{IPC}$$

$$E_{OT} = E_{IPC} \cup E_O$$

$$E_O = \{(o_p, o_1), (o_1, o_2), (o_2, o_3), \dots, (o_{p-1}, o_p)\}$$

$$\text{delay}(o_i, o_{i+1}) = 0 \text{ for } 1 \leq i < p$$

$$\text{delay}(o_p, o_1) = 1$$

Thus, an IPC graph can be modified by adding edges (the edges of  $E_O$ ) obtained from the ordering  $O$  to create the ordered transactions graph.

A closely related data structure is the *transaction partial order graph*  $G_{TPO}$ . The transaction partial order graph represents the minimum set of dependencies imposed among different processors by the communication actors of the IPC graph. These dependencies must be obeyed by any ordering of the communication operations. Under the assumption that the send and receive actors are serially ordered on each processor,  $G_{TPO}$  can be computed from  $G_{IPC}$  by first deleting all edges in  $G_{IPC}$  that have delays of one or more, and then deleting all of the computation actors [62]. However, since we wish to allow for the possibility of data transmission on multiple channels simultaneously we do not make this assumption, and we must modify the algorithm [8]. Figure 5.1 gives pseudo-code for our modified algorithm for generating  $G_{TPO}$ . The key difference with our algorithm is that  $G_{IPC}$  may now contain computation nodes with multiple predecessors and multiple successors. These nodes cannot be removed since this would require

**Algorithm 5.1:** GENERATE TPO GRAPH( $G_{IPC}$ )

**input:** IPC Graph  $G_{IPC}$

**output:** Transaction Partial Order TPO Graph

finished  $\leftarrow$  FALSE

**for** ( $\forall$  edges  $e \in G_{IPC}$ )

**do** { **if** ( $e$  is a feedback edge)  
    **then** { Delete  $e$

**while** (finished = FALSE)

    finished = TRUE  
    **for** ( $\forall$  nodes  $v \in G_{IPC}$ )  
        **if** ( $v$  is a computation node)  
            **if** ((indeg( $v$ ) = 0) OR (outdeg( $v$ ) = 0))  
                **then** { Delete  $v$   
                    finished = FALSE  
            **else if** (indeg( $v$ ) = 1)  
                **then** {  $p \leftarrow$  predecessor node of  $v$   
                    **for** ( $\forall$  successors  $s$  of  $v$ )  
                        **do** { Create edge ( $p, s$ )  
                        Delete  $v$   
                        finished = FALSE  
            **else if** (outdeg( $v$ ) = 1)  
                **then** {  $s \leftarrow$  successor node of  $v$   
                    **for** ( $\forall$  predecessors  $p$  of  $v$ )  
                        **do** { Create edge ( $p, s$ )  
                        Delete  $v$   
                        finished = FALSE

Figure 5.1: Pseudo-code for generating the TPO graph from the IPC graph.

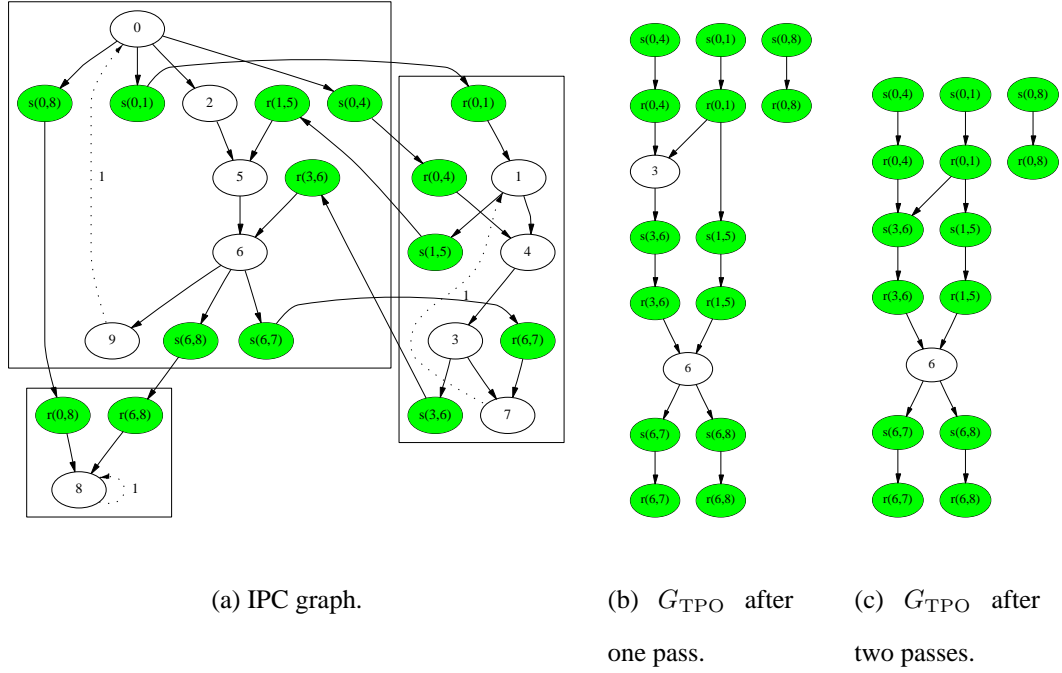


Figure 5.2: The TPO graph  $G_{TPO}$  derived from the IPC graph in (a) after 1 pass (b) and 2 passes (c) of the algorithm given in Figure 5.1.

imposing some additional dependencies on these nodes, and we want  $G_{TPO}$  to represent the minimal set of dependencies. In each pass of the algorithm, the graph is reduced by removing as many computation actors as possible. It terminates when no more computation actors can be removed. We will see in Section 5.2.2 that there are advantages to operating on the reduced TPO graph, since the search space of possible transaction orders can be exponential in the size of the graph.

Figure 5.2 shows an example of a how the transaction partial order graph is derived. The IPC graph is shown in 5.2(a). After one pass of the algorithm given in Figure 5.1, the TPO graph contains two computation actors (Figure 5.2(b)). The algorithm terminates after two passes with one computation actor remaining (Figure 5.2(b)).

As described earlier, when the ordered transaction strategy is implemented using

a hardware method such as micro-controller that imposes the linear order, there is no need for synchronization and contention for shared communication resources is also eliminated. Therefore, if the execution time estimates for the actors are accurate or are true worst-case values, then the maximum cycle mean (MCM) of the ordered transaction graph gives an accurate estimate or worst-case bound, respectively, of the iteration period of the associated application graph under the ordered transaction strategy. Such efficient, accurate performance assessment is useful for design space exploration in general, and it is especially useful when implementing applications that have real time constraints.

If interprocessor communication costs are negligible, an optimal transaction order can be computed in low polynomial time for a given self-timed schedule [102]. This method of deriving transaction orders is called the *Bellman-Ford Based* (BFB) method since it is based on applying the Bellman-Ford shortest path algorithm to an intermediate graph that is derived from the given self-timed schedule.

However, when IPC costs are not negligible, as is frequently and increasingly the case in practice, the problem of determining an optimal transaction order is NP-hard [62]. This intractability has been shown to hold both under iterative and non-iterative execution of application graphs. Thus, under nonzero IPC costs, we must resort to heuristics for efficient solutions. Furthermore, the polynomial-time BFB algorithm is no longer optimal, and alternative techniques to account for IPC costs are preferable.

In the presence of non-negligible communication costs, an efficient transaction order can be constructed with the help of the transaction partial order graph  $G_{\text{TPO}}$  described earlier. The *transaction partial order algorithm* is one systematic approach for using transaction partial order graphs to construct efficient orderings of communication operations. This algorithm proceeds by considering—one by one—each vertex of  $G_{\text{TPO}}$  that

has no input edges (vertices in the transaction partial order graph that have no input edges are called *ready* vertices) as a *candidate* to be scheduled next in the transaction order. Interprocessor edges are inserted from each candidate vertex to all other ready vertices in  $G_{IPC}$ , and the corresponding MCM is measured. The candidate whose corresponding MCM is the least when evaluated in this fashion is chosen as the next vertex in the ordered transaction, and deleted from  $G_{TPO}$ . This process is repeated until all communication actors have been scheduled into a linear ordering.

Khandelia [62] shows that the transaction partial order heuristic can improve the performance beyond what is achievable by a self-timed schedule, even if synchronization and arbitration costs are negligible compared to actor execution times. The performance benefit is achieved by strategic positioning of the communication operations in ways that do not result from the natural evolution of self-timed schedules.

## 5.2 WDM Ordered Transactions

In some applications, a shared electronic bus cannot handle the required communication traffic, even if this traffic is carefully optimized by using the transaction partial order heuristic. Moving to a faster shared medium such as optical ethernet may be a solution in some cases, since the interprocessor communication (IPC) is faster. However, we often cannot derive suitable solutions for highly parallel applications scheduled on multiple processors. In this case the shared nature of the interconnect becomes a bottleneck for the large amount of IPC required to effectively use all the processors. For these applications, alternative interconnection topologies are required.

One such alternative is to use multiple busses. Lee and Bier [67] describe how the ordered transaction strategy can be extended to utilize a hierarchy of multiple busses.

In Section 3.5.2 we introduced the WDMOT architecture utilizing fiber optics which is not a fully-connected topology (every processor pair having a dedicated channel), but can support multiple simultaneous communications on different wavelengths. The advantage of this architecture over the fully-connected topology is that the number of wavelengths required scales with  $N$  instead of  $N^2$ , where  $N$  is the number of processors. In this section we will discuss a heuristic for developing good processor orderings using this architecture, and compare the resulting system throughput with the throughput obtained using the transaction partial order algorithm for an electrical bus and with the throughput obtained in a fully connected system.

In the WDMOT architecture, we implement a protocol in which each processor is assigned a unique wavelength. We must ensure that two processors do not send to a given processor at the same time—i.e., there is possible contention at the (single) receiver of every processor. In order to accomplish this, we introduce a controller for every wavelength. This controller grants access to only one processor at a time. Figure 5.3 (repeated from Figure 3.10) depicts the architecture. Three fibers are shown—one to carry the data, one for the wavelength grant signals, and one for the wavelength release signals. The grant and release signals indicate that the wavelength is available (wavelength grant) or that a processor is finished using the wavelength (wavelength release or ack). The signal for wavelength  $\lambda_k$  being granted to processor  $p$  consists of an ID tag for processor  $p$  transmitted on  $\lambda_k$ . A portion of the grant signal is split off the grant fiber and distributed to each processor, where it is separated by wavelength. Processor  $p$  may transmit on  $\lambda_k$  if it receives its own ID tag on receiver  $k$ . This is shown in more detail in Figure 5.4 The controller for wavelength  $\lambda_j$  is responsible for ordering all communications to processor  $p_j$  (which receives data on  $\lambda_j$ ) so that only one processor is attempting to transmit on  $\lambda_j$  at a given time. In order to accomplish this, the wavelength controller



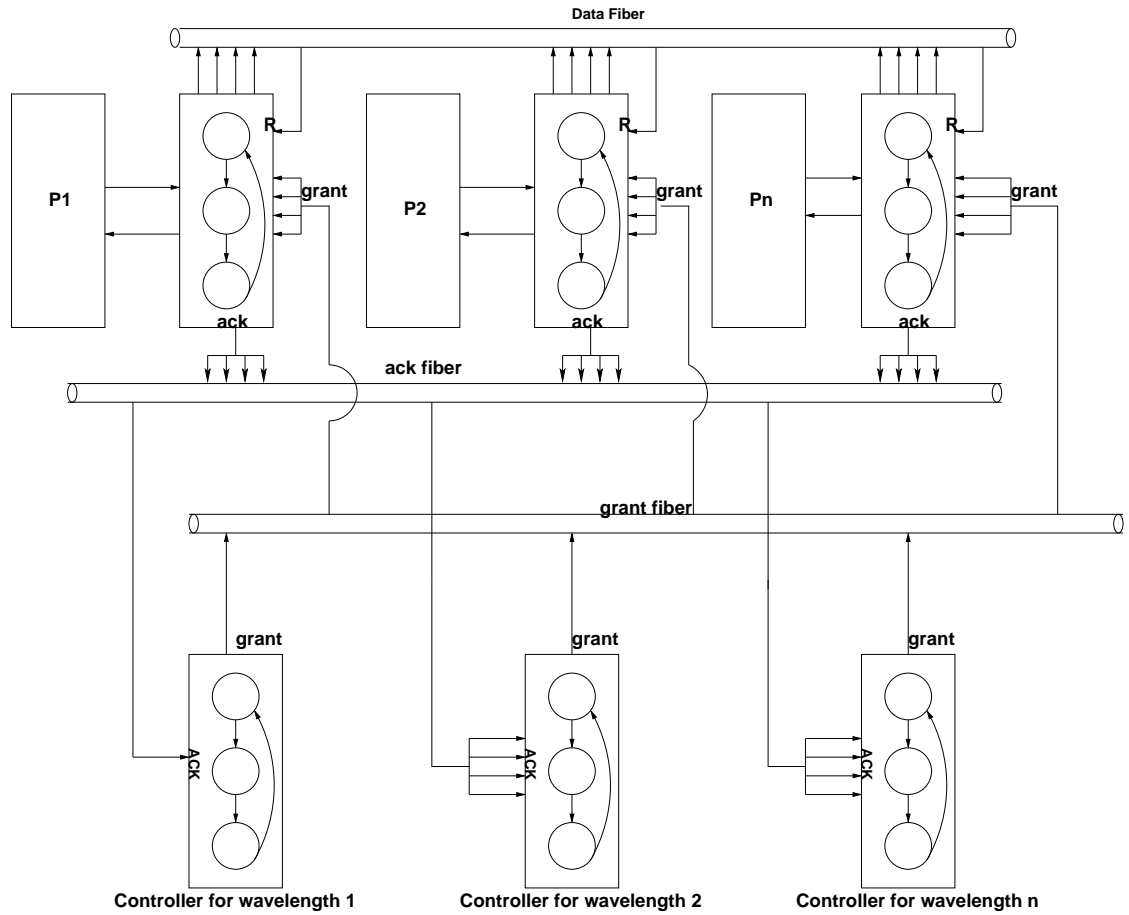
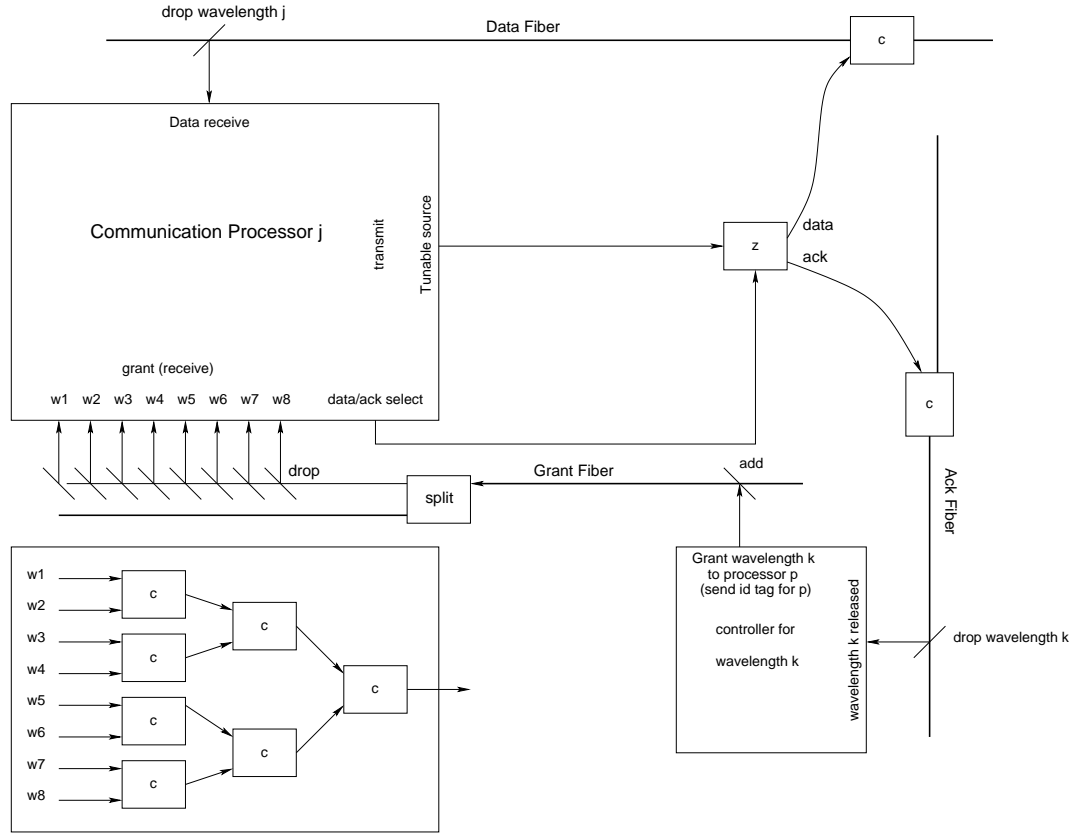


Figure 5.3: Architecture for wavelength ordered transactions (repeated from Figure 3.10).



Tunable source implemented by combining multiple discrete sources

Figure 5.4: One wavelength controller and one processor in the WDMOT architecture. The boxes marked  $c$  are wavelength combiners. The lower part of the figure shows how multiple discrete sources can be combined to form the tunable source.

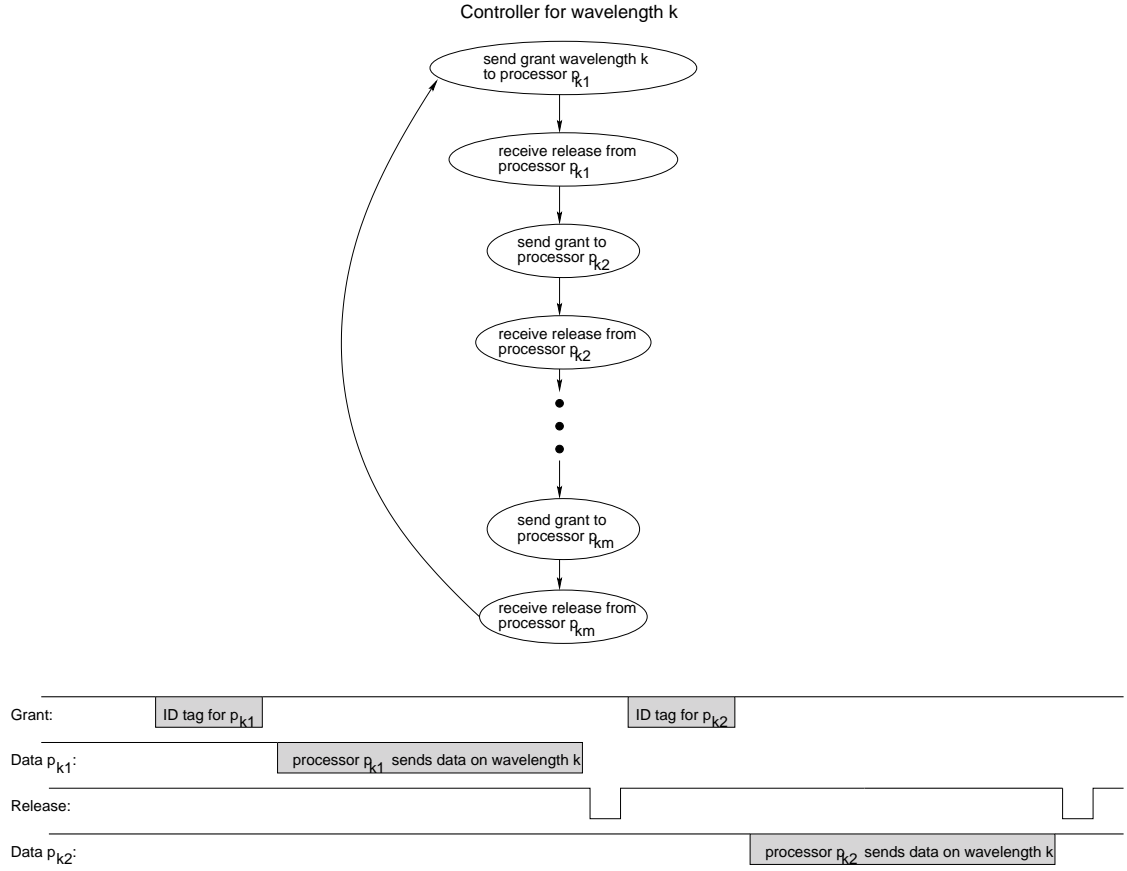


Figure 5.5: Controller state diagram and synchronization signals used in the WDMOT architecture.

must have a transmitter at the single wavelength  $\lambda_j$ , as well as a receiver at  $\lambda_j$ . Each processor  $p_j$  has a single receiver at  $\lambda_j$ , and a transmitter for each processor to which it will send data. This number may be limited by a *fan-out* constraint. Figure 5.5 shows the state diagram for a controller for wavelength  $\lambda_k$ . In Figure 5.5 there are  $m$  processors scheduled to transmit on  $\lambda_k$ , and they are ordered  $[p_{k1}, p_{k2}, \dots, p_{km}]$ . Since the order of grants to  $\lambda_k$  is enforced by the controller, only one processor transmits on  $\lambda_k$  at a given time.

In Figure 5.4 we show a tunable source being used in conjunction with each pro-

cessor. Most commercially available tunable sources today use some type of microelectromechanical (MEMS) switching element, and require times of the order of milliseconds to tune between different wavelengths. We can consider this tuning time as an addition to the interprocessor communication time, and would like switching times on the order of several clock cycles, which is on the order of nanoseconds. One way to accomplish this (although not strictly a ‘tunable’ source) is to simply use multiple discrete sources, which can be selected electrically, and combine them together. This is shown in the lower portion of Figure 5.4. Several groups are currently developing fast (Gb/s) tunable wavelength converters for optical switching applications (e.g. see [75, 89]), although the number of output wavelengths demonstrated has been small. For example, Mašanović et al. recently reported a tuning range of 22nm for an ImP tunable laser and wavelength converter [75]. These devices may one day be suitable for the WDMOT architecture.

Using the WDMOT architecture, any given interconnect topology that respects the fanout constraints can be implemented. We show in Section 6.2 that the optimal interconnect topology for a given application is often very irregular. We will explain in Chapter 7 how to synthesize an optimal interconnect topology for a given application with these fanout constraints.

### 5.2.1 Optical Components

The WDMOT architecture can be implemented with components developed for the telecommunications market. The optical add/drop multiplexer (OADM) is a basic building block of many optical systems where signals with arbitrary wavelengths must be multiplexed to, or demultiplexed from, wavelength multiplexed signals. Figure 5.6 [55] depicts the basic configuration of an OADM using a dielectric thin film filter. OADMs

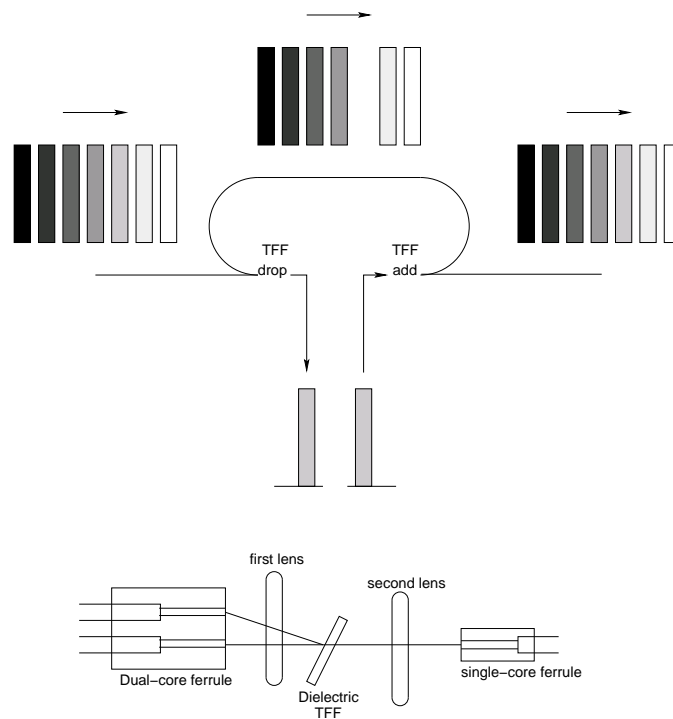


Figure 5.6: Optical add/drop multiplexer utilizing a dielectric thin film filter.

can be used at each processor to extract the correct wavelength for incoming data, to separate the grant signals from the controllers, to multiplex outgoing data onto the data fiber, and to multiplex the acknowledge signal onto the acknowledge fiber. At each controller, OADM's can be used to add the grant signal onto the grant fiber, and to drop the release signal from the acknowledge fiber.

### 5.2.2 Transaction Ordering

We define an ordered transaction graph  $G_{\text{WDMOT}}$  in a similar manner to Section 5.1.3. Let  $\epsilon_p$  be the set of communication edges in  $G_{\text{IPC}}$  whose target nodes are scheduled on processor  $p$ , and  $\eta_p$  be the set of nodes that are sources for edges  $\epsilon_p$ . In order to ensure that no two processors attempt to transmit on the same wavelength at the same time, we must determine a transaction ordering  $O_p$  for the nodes in  $\eta_p$  for  $p \in [1 \dots N]$  where  $N$  is the number of processors. Then

$$\Upsilon(G_{\text{IPC}}, O_1, \dots, O_N) = G_{\text{WDMOT}} = (V_{\text{WDMOT}}, E_{\text{WDMOT}}),$$

where

$$V_{\text{WDMOT}} = V_{\text{IPC}}, \text{ and}$$

$$E_{\text{WDMOT}} = E_{\text{IPC}} \cup E_{O_1} \cup E_{O_2} \cup \dots \cup E_{O_N}.$$

Figure 5.7 shows ordered transaction graphs for both electrical shared bus and WDMOT architecture. For the electrical shared bus, all communications must be ordered and four additional edges must be added to  $G_{\text{IPC}}$ . For the WDMOT architecture only two additional edges must be added to  $G_{\text{IPC}}$ —nodes A and F which both send to processor 2 must be ordered, and nodes G and J which both send to processor 4 must be ordered. Note that since the iteration cycle time (reciprocal of the throughput) of the system is determined by the MCM of the ordered transaction graph, and since adding edges to a

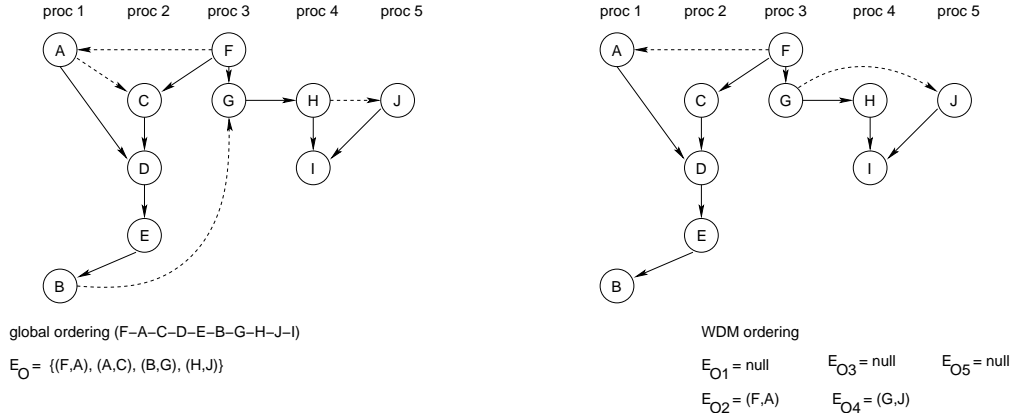


Figure 5.7: Comparison of ordered transaction graphs for shared bus (left) and WDM architecture (right). Transaction order edges are shown dashed.

graph cannot decrease the MCM, the throughput of the WDMOT architecture cannot be less than that of the electrical shared bus architecture. Next we address the question of determining the orderings  $O_p$  for each wavelength.

We first note that for the ordering to be correct, we should not introduce any zero-delay cycles into the IPC graph. Such cycles would create deadlock in the system. In Figure 5.7 for example, an ordering beginning with  $F \rightarrow C \rightarrow D \rightarrow E \rightarrow A$ , placing  $E$  before  $A$ , would add the edge  $(E, A)$  to  $G_{IPC}$  and create the cycle  $A \rightarrow C \rightarrow D \rightarrow E \rightarrow A$ . In other words, the transaction ordering should be a topological sort of the directed acyclic graph resulting from removing the feedback edges from  $G_{IPC}$ . Equivalently, since  $G_{TPO}$  preserves all the dependencies in  $G_{IPC}$ , the transaction ordering must be a topological sort of  $G_{TPO}$ , and our goal is then to find the best topological sort. Unfortunately, the number of possible topological sorts can be exponential in the size of the graph. For example, for a complete bipartite graph with  $2n$  nodes, there are  $(n!)^2$  different topological sorts. We therefore see that we can reduce the search space substantially by reducing the IPC graph and operating on the TPO graph.

**Algorithm 5.1:** CHOOSE COMMUNICATON ACTOR( $G_{IPC}, \text{readyList}$ 

)

**input:** ipc graph  $G_{IPC}$ **input:** list of actors readyList**output:** communication actor  $\nu$ **if** readyList.size()  $\equiv 1$     **then**  $\{\nu \leftarrow \text{readyList.head}()\}$ **for**  $x \in \text{readyList}$ 

do	{	do	{	<b>for</b> $y \in \text{readyList}$
				<b>if</b> $x \neq y$
				<b>then</b> $\{e = G_{IPC}.\text{addege}(x, y)$
				temp.add( $e$ )
				criteria[ $x$ ] $\leftarrow \text{MCM}(G_{IPC})$
				<b>for</b> $e \in \text{temp}$
				<b>do</b> $\{G_{IPC}.\text{delete}(e)$
				$\nu \leftarrow \min(\text{criteria}[x])$

Figure 5.8: Function to choose the next communication actor in the transaction order [62].

We will see in Section 5.2.3 that a random topological sort produces relatively poor results, and we must derive heuristics to guide the sort. We use a modification of the transaction partial order heuristic [62]. In this heuristic, edges are added between communication nodes (actors) in  $G_{IPC}$  that are contending for the bus, and the MCM of the modified  $G_{IPC}$  is measured. Actors whose corresponding MCMs are better are scheduled earlier in the transaction order, as discussed in Section 5.1.3. The algorithm for choosing the next communication actor to schedule is given in Figure 5.8 while the transaction partial order heuristic is given in Figure 5.9. For the WDMOT architecture, we must determine an ordering for each wavelength controller. In order to do this, we first determine a global ordering of the communication actors using the TPO heuristic,



**Algorithm 5.2:** TPO HEURISTIC(  
 $G_{\text{IPC}}, \text{transactionOrder}$   
 $)$

**input:** IPC graph  $G_{\text{IPC}}$   
**output:** linear list of communication actors transactionOrder

compute  $G_{\text{TPO}}$  from  $G_{\text{IPC}}$   
**for**  $\nu \in G_{\text{IPC}}$   
     $\text{mark}[\nu] \leftarrow \text{FALSE}$   
    **do**  $\left\{ \begin{array}{l} \text{if } \text{indegree}(\nu) = 0 \\ \text{then } \{\text{readyList.append}(\nu)\} \end{array} \right.$

$\text{complete} \leftarrow \text{FALSE}$   
 $\text{first} \leftarrow \text{TRUE}$   
**while** ( $\text{complete} \neq \text{TRUE}$ )  
     $\nu \leftarrow \text{CHOOSE-COMMUNICATION-ACTOR}(G_{\text{IPC}}, G_{\text{TPO}}, \text{readyList})$   
     $\text{mark}[\nu] \leftarrow \text{TRUE}$   
     $\text{transactionOrder.append}(\nu)$   
    **if** ( $\text{first}$ )  
        **then**  $\{\text{first} \leftarrow \text{FALSE}\}$   
    **else**  $\{G_{\text{IPC}}.\text{addege}(w, \nu)$   
         $w \leftarrow \nu$   
    **do**  $\left\{ \begin{array}{l} \text{for } u \in (\nu, u) \in E \\ \text{do } \left\{ \begin{array}{l} \text{flag} \leftarrow \text{TRUE} \\ \text{for } s \in (s, u) \in E \\ \text{do } \left\{ \begin{array}{l} \text{if } (\text{mark}(s) = \text{FALSE}) \\ \text{then } \{\text{flag} \leftarrow \text{FALSE}\} \end{array} \right. \\ \text{if } (\text{flag}) \\ \text{then } \{\text{readyList.append}(u)\} \end{array} \right. \\ \text{if } (\text{readyList.empty}() = \text{TRUE}) \\ \text{then } \{\text{complete} \leftarrow \text{TRUE}\} \end{array} \right.$

Figure 5.9: TPO heuristic [62].

**Algorithm 5.3:** WDM TPO HEURISTIC(  
 $G_{\text{IPC}}, \Omega$   
)  
**input:** IPC graph  $G_{\text{IPC}}$   
**output:** linear list of communication actors  $\Omega_j$  for each wavelength  $j$

CALL HeuristicTPO( $G_{\text{IPC}}$ , transactionOrder)  
**for**  $p \in [1 \dots N]$   
  **do**  $\left\{ \begin{array}{l} \text{for edges } e \in G_{\text{IPC}} \\ \quad \text{do } \left\{ \begin{array}{l} \text{if } \text{proc}(\text{target}(e)) = p \\ \quad \text{then } \{ \epsilon_p \leftarrow \epsilon_p \cup e \} \end{array} \right. \\ \text{for edges } e \in \epsilon_p \\ \quad \text{do } \{ \eta_p \leftarrow \text{source}(e) \} \end{array} \right.$   
 $\Omega_p \leftarrow \text{SORT}(\eta_p \text{ using transactionOrder})$

Figure 5.10: WDM ordered transactions algorithm.

and then use this ordering to sort the actors in each  $\eta_p$ . Pseudocode for the algorithm is given in Figure 5.10.

### 5.2.3 Experiments

We ran the WDM ordered transactions algorithms on a set of randomly generated graphs. These graphs were generated using a modified version of Sih's method [96] which produces graphs with a regular structure that resembles many DSP applications. We modified Sih's algorithm to insure that the random connections do not introduce cycles, and added a fanout parameter which controls the amount of parallelism in the graph. Pseudocode for the random graph generation algorithm is shown in the Appendix. An example of a random graph generated using this algorithm is shown in Figure 5.11.

Figure 5.12 compares the WDM ordered transaction heuristic (using the fiber-based WDMOT architecture) to the TPO heuristic (using a shared bus architecture) and a trans-

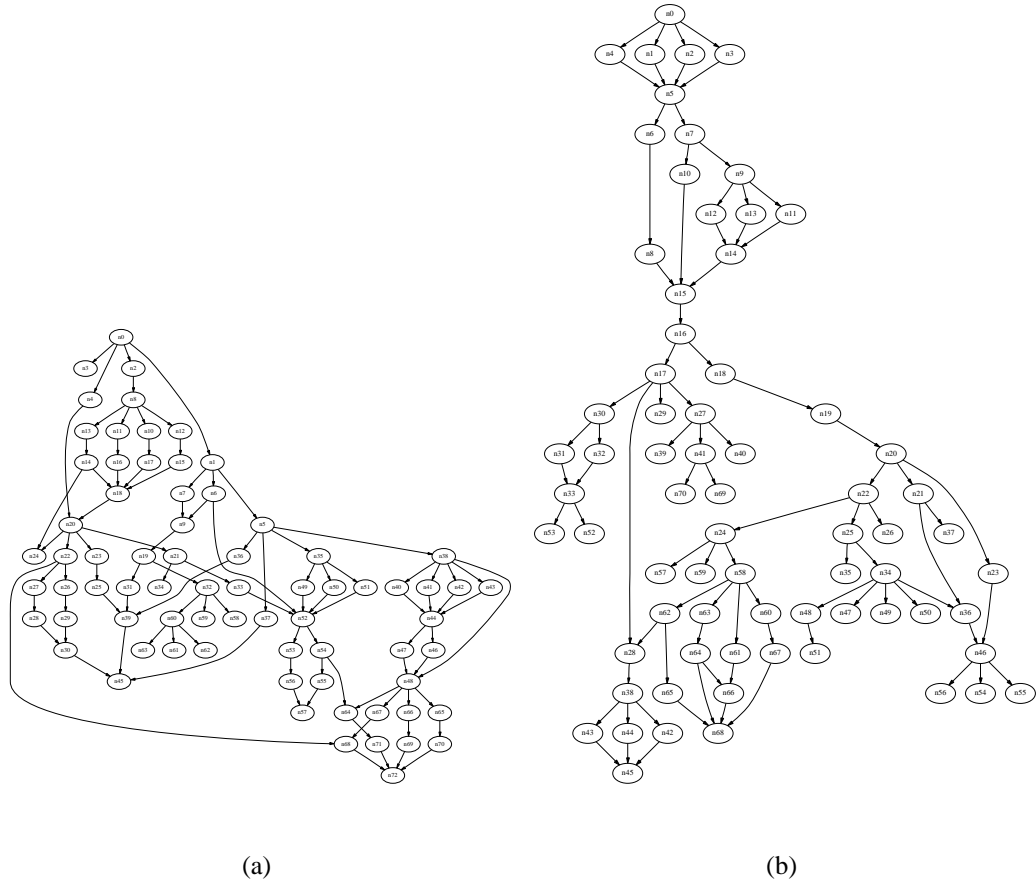


Figure 5.11: Two examples of random graphs generated using a modification of Sih's algorithm [96] with 70 nodes and fanout 5.

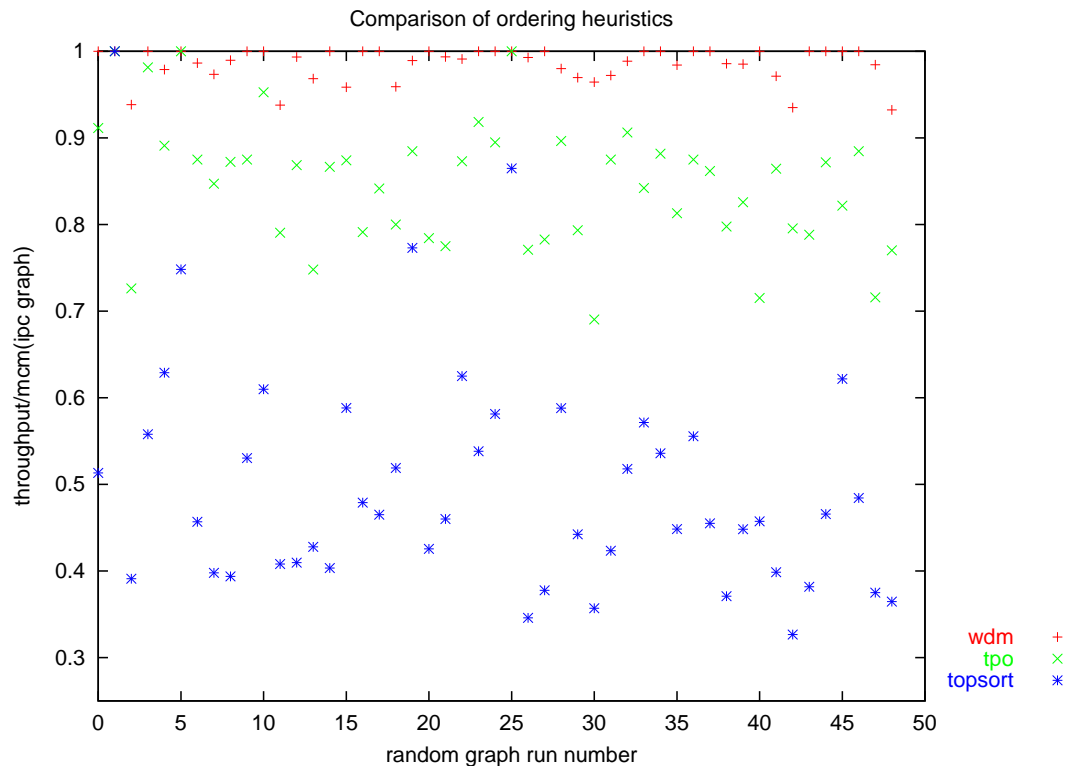


Figure 5.12: Comparison of TPO heuristic, WDMOT heuristic, and random topological sort ordering for a set of random graphs.

action order based on a random topological sort (using a shared bus architecture) for a set of randomly generated graphs. The y-axis in Figure 5.12 is the ratio of throughput to the maximum possible throughput (the reciprocal of the MCM of  $G_{IPC}$ ) (the *throughput ratio*) that would be obtained for a full crossbar interconnect. In Figure 5.12 the ratio of average execution time for the computation actors to the average IPC communication time (the *computation ratio*) was 2.5. We observe that the throughput using the WDMOT architecture is usually very close to this maximum in these graphs, and generally better than that for the shared bus. We also observe a significant improvement for the TPO heuristic over the topological sort ordering. For purposes of comparison we assume the same communication times for both optical and electrical busses in this experiment. In practice the communication times for the optical bus would be lower, which would increase the relative improvement of the WDMOT results.

In Figure 5.13 we plot the average, over 50 random graphs, of the throughput ratio as the computation ratio is varied. We see that the WDMOT architecture produces throughput very close to the theoretical maximum for this size graph ( $L = 8$  in Sih's algorithm and 4 processors). Also, the relative performance of WDMOT to the shared bus increases as the communication overhead increases (lower computation ratio on the x-axis). The random topological sort performs significantly worse. Also, there is no improvement as the communication overhead decreases. This is because the random topological sort is imposing an inefficient ordering of the computation nodes as well as the communication nodes, and this effect is dominant.

In Figure 5.14 we plot the average, over 50 random graphs, of the throughput ratio as the number of processors is varied. Again we see that the WDMOT architecture performs close to the theoretical maximum. We also observe that its performance is less sensitive to the number of processors.

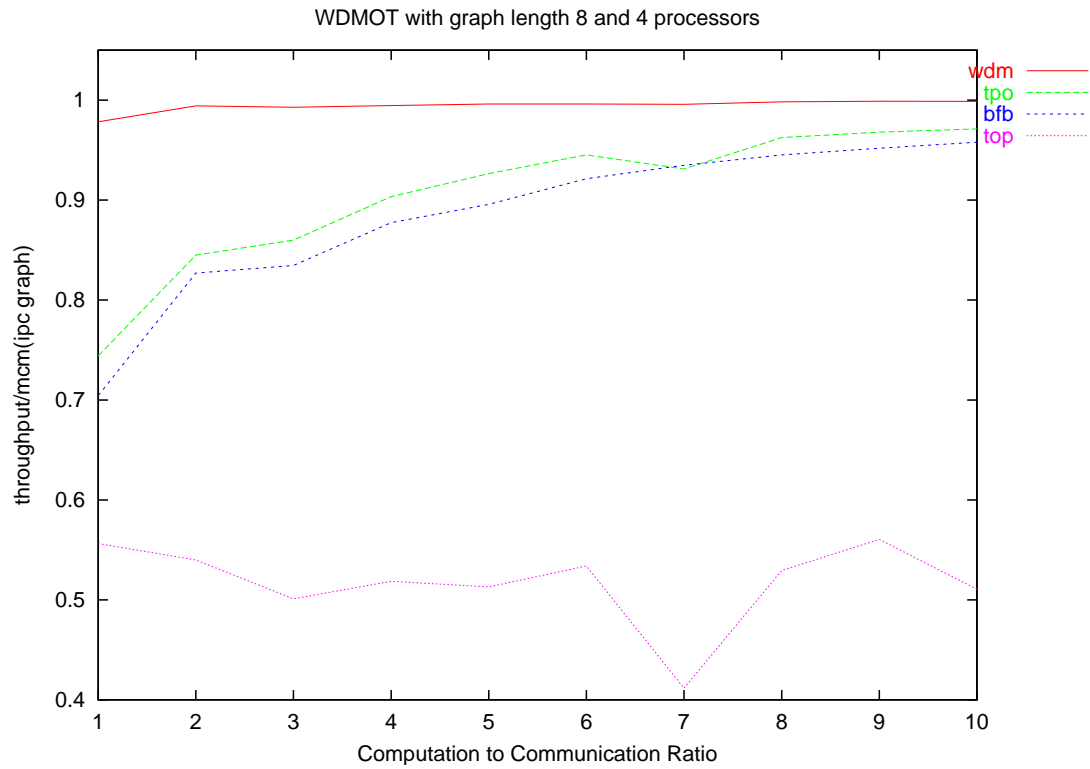


Figure 5.13: Comparison of TPO heuristic, WDMOT heuristic, and random topological sort ordering vs. computation ratio. Each point is the average of 50 random graphs. The graph length was 8 and the application was scheduled on 4 processors.

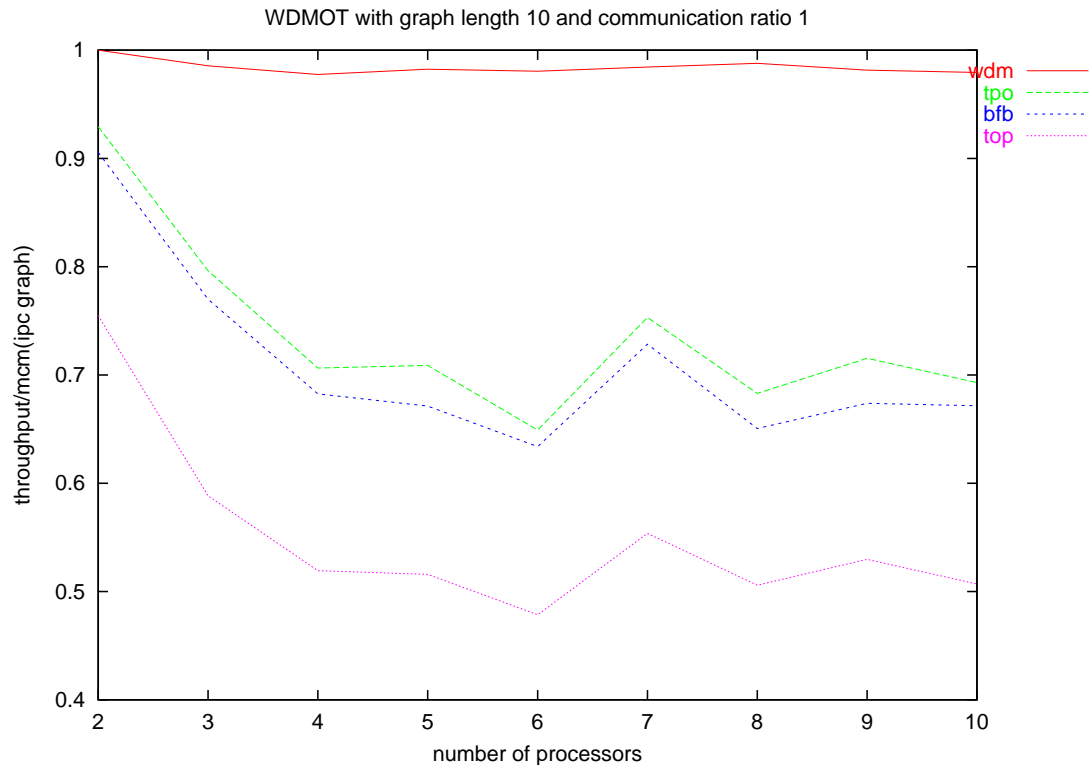


Figure 5.14: Comparison of TPO heuristic, WDMOT heuristic, and random topological sort ordering vs. number of processors. Each point is the average of 50 random graphs. The graph length was 10 and the communication ratio was fixed at 1.

## Chapter 6

### Scheduling for Arbitrarily Connected Systems

Scheduling for multiprocessor systems was introduced in Chapter 2. A vast range of scheduling techniques for task graphs has been developed (e.g. see [101] for a review of several representative approaches); however, these techniques typically assume a fixed communication network, and do not systematically take connectivity constraints into account. By connectivity constraints, we mean the inability of certain pairs of processors to communicate with each other. Such constraints are desirable to impose in optically connected multiprocessors because the power consumption of communication is relatively independent of distance, and largely dependent instead on the number of electrical-to-optical conversions that must be performed (this will be discussed further in Section 6.2).

Thus, it is advantageous to configure multiprocessor schedules in such a way that multi-hop communication is avoided, or limited to some maximum number of hops per communication operation, and the relative abundance of communication links is used instead to achieve the required communication flexibility. However, such connectivity constraints can cause list scheduling techniques, and related methods to deadlock. One contribution of this thesis is to develop a general framework for extending arbitrary



list scheduling approaches to avoid deadlock, and operate efficiently in the presence of connectivity constraints. We will apply this framework to jointly streamline the communication network and task graph mapping for a given application in Chapter 7. This framework can be used both for minimum-cost dedicated implementations, and for re-configurable networks, where the goal is to save power consumption by activating a minimal subset of available laser-detector pairs.

## 6.1 Implications of Increased Connectivity

It has been shown that optical interconnects can provide a higher degree of connectivity than electrical interconnects. For example, Li [71] claims that using technology available in the year 2000, interconnects of up to 1000x1000 I/O elements per square centimeter can be achieved. In this thesis we explore the implications of various levels of connectivity for multiprocessor systems, and interconnection networks that can make use of the unique properties of optical interconnects.

One consequence of increasing levels of connectivity between processors is that it is easier for mapping algorithms to find good solutions. We show here that the quality and number of solutions found by a probabilistic search algorithm is a strong function of the level of connectivity. Connectivity can be defined in the following way for a system with  $N$  processors  $\{P_1, P_2, \dots, P_N\}$ : Let  $P_X \rightarrow P_Y \equiv \text{TRUE}$  iff there exists a direct communication link from  $P_X$  to  $P_Y$ . *Directional* connectivity is defined as

$$\sum_X \sum_{Y \neq X} \text{Ivalue}(P_X \rightarrow P_Y)$$

*Directionless* connectivity is defined as

$$\sum_X \sum_{Y < X} \text{Ivalue}((P_X \rightarrow P_Y) \vee (P_Y \rightarrow P_X))$$

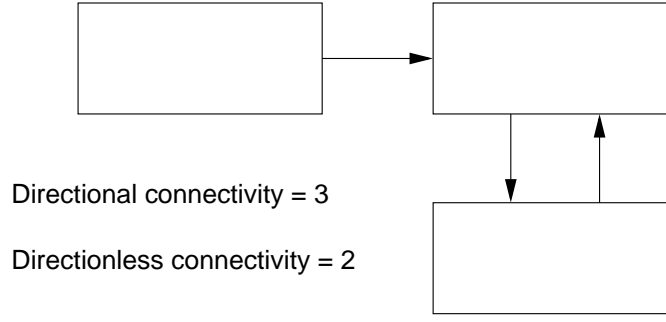


Figure 6.1: Example of directional and directionless connectivity.

where  $\text{Ivalue}(\text{TRUE}) = 1$  and  $\text{Ivalue}(\text{FALSE}) = 0$ . An example is shown in Figure 6.1.

### 6.1.1 Topology Graph

We define a topology graph  $T(\Phi, L)$  such that the nodes of  $T$  correspond to the “processors”  $\Phi$  in the architecture and the edges  $L$  in  $T$  correspond to direct physical communication links between the processors. We define the set of all processors as  $\Phi$  and label the processors  $\{p_1, p_2, \dots, p_{|\Phi|}\}$ . Then  $T$  contains an edge  $(p_i, p_j)$  iff the interconnection network provides a direct (single-hop) communication link from  $p_i$  to  $p_j$ . If  $l$  is an edge in  $T$ , we say that  $\text{src}(l)$  is the *source* node of  $l$ ;  $\text{snk}(l)$  is the *sink* node of  $l$ ;  $l$  is *directed* from  $\text{src}(l)$  to  $\text{snk}(l)$ ;  $l$  is an *output edge* of  $\text{src}(l)$ ; and  $l$  is an *input edge* of  $\text{snk}(l)$ . We denote the *degree* of a processor by the number of incident (physical) communication links. The degree of a node  $\nu$  in  $T$  is equal to the sum of the number of input and output edges of  $\nu$ . For example, each processor in a fully-connected system with  $|\Phi|$  processors, has degree  $2(|\Phi| - 1)$  (two links—one incoming and one outgoing—to each other processor). Furthermore, a *path* in  $T(\Phi, L)$  is a nonempty sequence  $l_1, l_2, l_3, \dots \in L$  such that  $\text{snk}(l_1) = \text{src}(l_2), \text{snk}(l_2) = \text{src}(l_3), \dots$  whose *path length* equals the num-

ber of edges in the sequence.  $T$  is said to be *strongly connected* if for each pair of distinct nodes  $(p_1, p_2)$  there is a path directed from  $p_1$  to  $p_2$  and there is a path directed from  $p_2$  to  $p_1$ .

### 6.1.2 Effect of Connectivity on a Simple Mapping Algorithm

We begin by experimenting with a very simple mapping algorithm in order to observe the effects of different levels of connectivity. In this experiment, a synthetic aperture radar application with 60 tasks was mapped onto a 9 processor heterogeneous architecture with the purpose of studying the resulting connectivity patterns. The goal of the mapping algorithm was simply to determine which tasks should be assigned to which processors, not the relative ordering of the tasks on the processors, or the effects of inter-processor communication or iterative execution. With these numbers, the search space consists of approximately  $2 \cdot 10^{57}$  distinct mappings. A genetic algorithm was used to explore this space. Performance was measured as a function of connectivity constraint (i.e., an upper limit on allowable connectivity), with 10 trials for each connectivity constraint point. Figure 6.2 shows the number of valid solutions found by the genetic algorithm vs. connectivity. Figure 6.3 shows the best throughput obtained as a function of the connectivity constraint. It can be seen that both the number of valid solutions and the quality of these solutions increase with increased amounts of connectivity.

## 6.2 Connection Topologies

Electrically connected systems generally have a regular interconnection pattern, due to the physical constraints imposed by two-dimensional circuit board layout. Some examples include ring, mesh, bus, and hypercube interconnect topologies. Using these

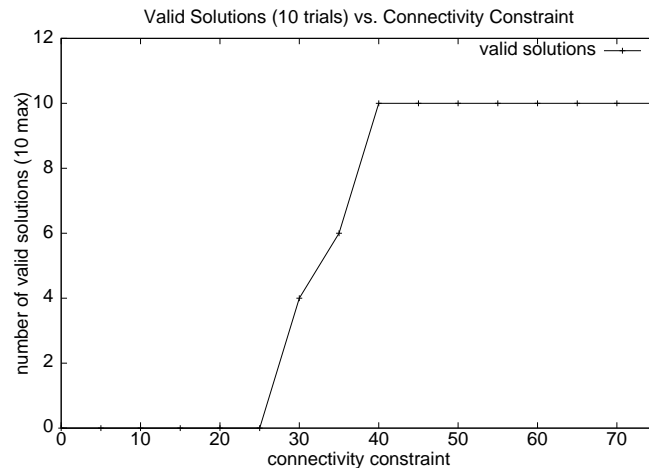


Figure 6.2: Impact of Connectivity on Search Efficiency.

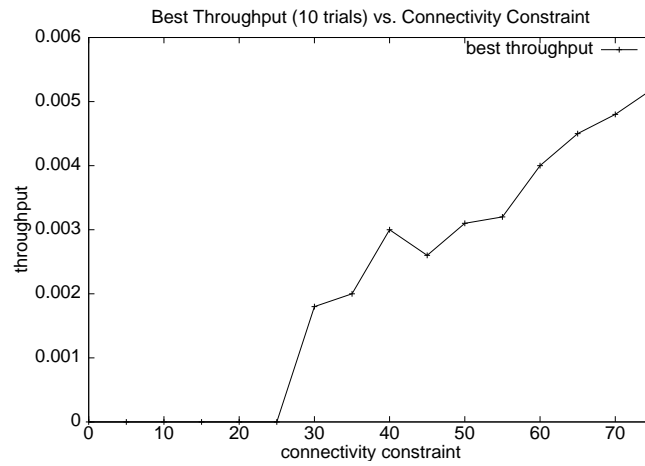


Figure 6.3: Impact of Connectivity on Performance.

topologies, communication between remote processors requires multiple hops, which increases both latency and power, and increases contention throughout the network.

In contrast, optically connected multiprocessors, particularly those utilizing free space optics and three dimensions, are free to utilize arbitrarily irregular interconnection networks. Once the signal is in the optical domain, there is very little attenuation, so the energy required to transmit a unit of data is essentially independent of distance. The required energy instead is a function of the number of electrical-to-optical conversions that must be performed [63], which in turn is determined by the number of hops. Furthermore, due to the flexibility of the communication medium, it is generally possible to avoid multi-hop communication operations by simply activating direct communication channels between the source and destination processors. Together, these properties make it desirable to limit the number of hops per communication operation when exploring configurations (interconnection patterns and task graph mappings) for an optically connected, embedded multiprocessor.

Irregular interconnection patterns arise naturally when scheduling task graphs under the restriction of single-hop communication. A simple example of an irregular interconnection network is shown in Figure 6.4. Given four processors and four bidirectional links, there are two possible topologies shown in Figure 6.4(a) and (b). Topology (a) has a regular interconnection pattern, with each processor connected to two others. Topology (b) is irregular, with one processor having degree three, one processor having degree one, and the others having degree two. Topology (b) allows a single-hop schedule, since all required communication can take place with only one hop. In topology (a), two hops are required for communication from task  $A$  to task  $D$  and from  $D$  to  $E$ .

Task graph scheduling algorithms generally produce schedules that require an irregular interconnect topology for single-hop communication. For example, Figure 6.5

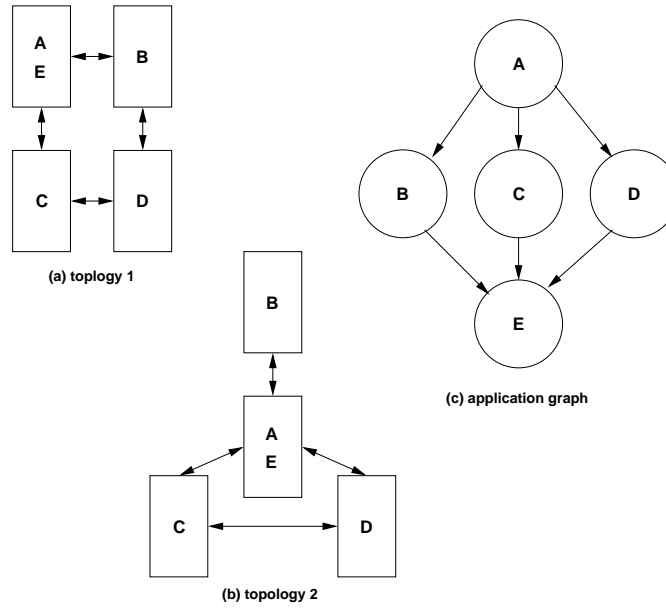


Figure 6.4: (a)-(b) Two possible topologies with four processors and four bidirectional links. (c) Application graph.

shows the application graph for an FFT application [56]. This application was scheduled on eight processors using the DLS algorithm [96], with no constraints made on interconnect placement. Figure 6.6 shows the topology required to operate the resulting schedule using only single-hop communication operations. There are 14 directional links out of a possible 56 for a fully connected system (the ratio of these two numbers gives a measure of the average connectivity of each processor).

If we denote the *degree* of a processor by the number of incident (physical) communication links, each processor in a fully-connected  $n$  processor system, for example, has degree  $2(n - 1)$  (two links—one incoming and one outgoing—for each processor). In an arbitrary network, the relative variation in the degrees among different processors gives a measure of the level of irregularity of the associated interconnection pattern. For example, in the mapping of Figure 6.6, processors 0 and 6 have degree six, while pro-

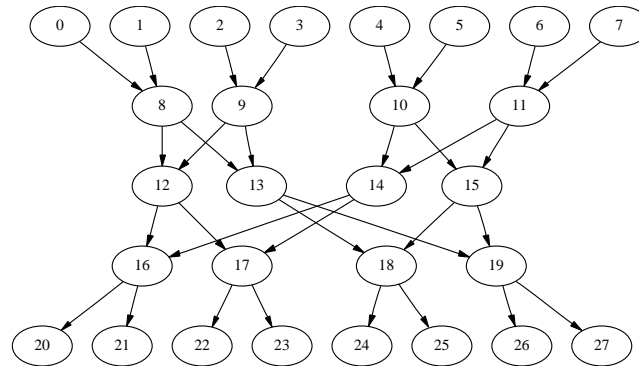


Figure 6.5: FFT1 application graph.

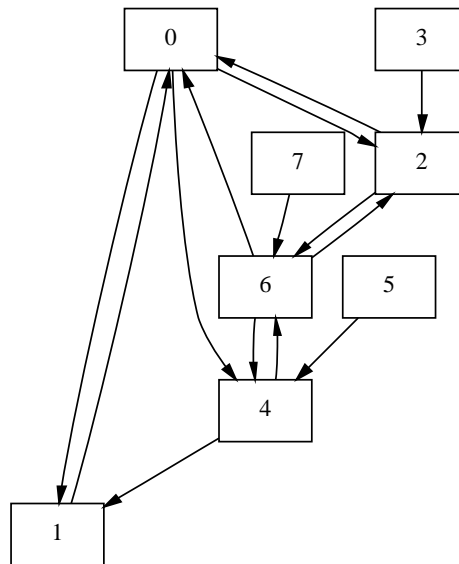


Figure 6.6: Single-hop processor topology for DLS schedule of FFT1 application.

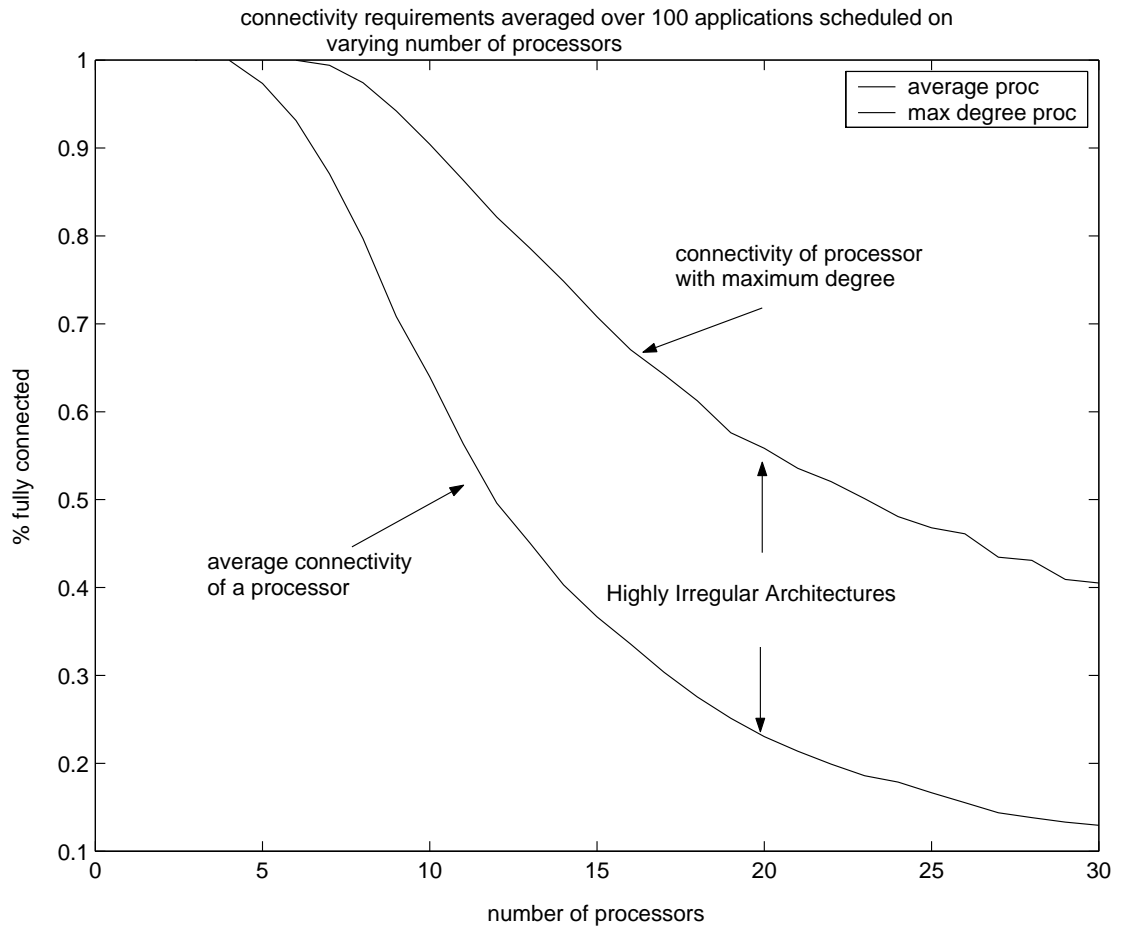


Figure 6.7: Connectivity requirements of 100 benchmark applications.

processors 3 and 5 have degree one. This trend of highly irregular connection requirements occurs over a wide variety of task graph structures. To illustrate this, Figure 6.7 plots the average of these measures over 100 real and synthetic benchmark application graphs when scheduled on different numbers of processors. The synthetic benchmarks used in these experiments were generated using the graph generation techniques of Sih [96], which are designed to construct task graphs that resemble the dataflow structures found in DSP applications.

As motivated earlier, when developing automated mapping tools for optically con-



nected systems, we have several design constraints. It is desirable to map the application onto the architecture without requiring multi-hop communication, while satisfying constraints on system throughput and latency. We also have limits on the maximum I/O fanout and degree of a single processor. In order to conserve space and power, we would also like to minimize the total number of communication links.

### 6.3 Connectivity and Scheduling Flexibility

Due to the desirability of single-hop communications in optically interconnected multiprocessors, as motivated in Sections 6.1 and 6.2, it is important during co-design to employ scheduling techniques that carefully take into account the connectivity of candidate interconnection patterns. In systems that are not fully connected, the consequence of single-hop communication is that each processor  $p$  can only send data to some subset  $\chi(p)$  of the set of all processors  $\Phi$ , and only receive data from a subset  $\Omega(p)$  of  $\Phi$ .

If these constraints are not considered, deadlock can easily occur during the scheduling process. Consider an application graph  $G$ , two tasks  $\nu_1$  and  $\nu_2$  in  $G$  that have been scheduled on processors  $p_1$  and  $p_2$ , respectively, and a third task  $\nu_3$  that receives data from  $\nu_1$  and  $\nu_2$ . Then if  $\chi_1(\nu_1) \cap \chi_2(\nu_2) = \emptyset$ , the scheduler is deadlocked.

We define a *feasible set* of processors  $\Psi[\nu]$  for a task  $\nu$  as the largest subset of  $\Phi$  on which  $\nu$  can be scheduled without deadlock. We would like to have an algorithm to determine the feasible set of processors  $\Psi[\nu]$  for all  $\nu \in G$ . In general, a constraint imposed on one task (scheduling it on a processor) may cause  $\Psi[\nu]$  to be updated for all  $\nu \in G$ . This update consists of choosing a subset of the set  $\Psi[\nu]$  that existed before the constraint—new members are never added.

We define the *communication flexibility* (or simply *flexibility* for short) of the system

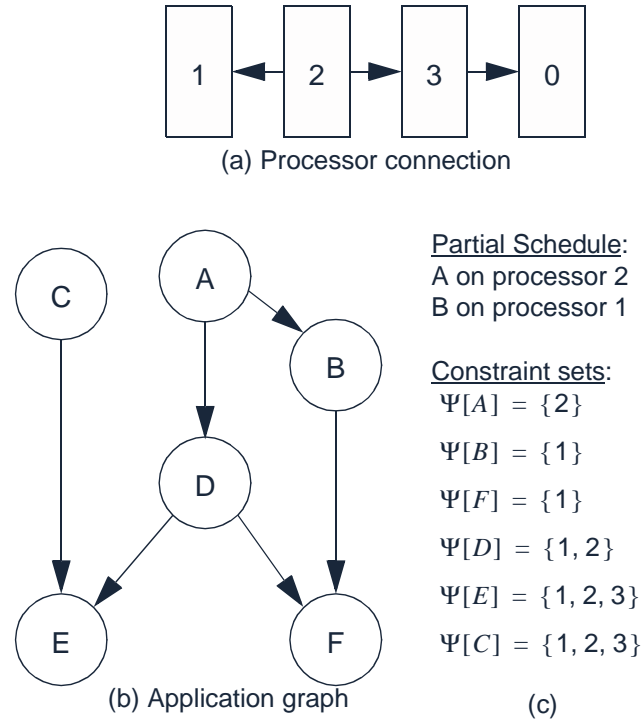


Figure 6.8: Example requiring constraint information propagating both forward and backward.

at any point during the scheduling process as the sum of the sizes of the sets  $\Psi[\nu]$  for all  $\nu \in G$ . The flexibility gives some measure of the degree of constraint imposed on all tasks by a given scheduling move. Figure 6.8 depicts a simple example of an application graph with six tasks being scheduled on four processors. A partial schedule is shown in Figure 6.8(c). Scheduling task  $B$  in Figure 6.8(b) has an effect on tasks  $C$ ,  $D$ ,  $E$ , and  $F$ . Figure 6.8(c) shows the constraint sets  $\Psi$  for each task after scheduling  $B$  on processor 1. The flexibility at this point is equal to 11. If  $B$  had been scheduled on processor 2, the flexibility would be 16. This example also demonstrates the potential for deadlock. After task  $B$  is scheduled on processor 1, processor 0 becomes infeasible for task  $C$ , since scheduling task  $C$  on processor 0 confines task  $E$  to processor 0. Task

**Algorithm 6.1:**  $Rf^1(S)$ 

**input:** set of processors  $S \in \Phi$   
 $\Phi$  is the set of all processors  
**output:** set of processors  $R$  that can be reached from  $S$  in zero or one hop

```

 $R \leftarrow S$ 
for all  $p \in S$ 
  do  $\left\{ \begin{array}{l} \text{for all } z \in \Phi \\ \text{do } \left\{ \begin{array}{l} \text{if } C(p, z) \equiv \text{TRUE} \\ \text{then } \{R \leftarrow R \cup \{z\}\} \end{array} \right. \end{array} \right.$ 
return  $(R)$ 

```

Figure 6.9: Function  $Rf^1(S)$ .**Algorithm 6.2:**  $Rb^1(S)$ 

**input:** set of processors  $S \in \Phi$   
 $\Phi$  is the set of all processors  
**output:** set of processors  $R$  that can reach at least one element in  $S$  with zero or one hop

```

 $R \leftarrow S$ 
for all  $p \in S$ 
  do  $\left\{ \begin{array}{l} \text{for all } z \in \Phi \\ \text{do } \left\{ \begin{array}{l} \text{if } C(z, p) \equiv \text{TRUE} \\ \text{then } \{R \leftarrow R \cup \{z\}\} \end{array} \right. \end{array} \right.$ 
return  $(R)$ 

```

Figure 6.10: Function  $Rb^1(S)$ .

$F$  is confined to processor 1 since  $B$  is scheduled on 1. Task  $D$  sends data to both  $E$  and  $F$ , and there is no processor which can communicate with both processors 0 and 1 in a single hop. Existing scheduling algorithms are not designed to detect this deadlock condition. Avoiding these deadlock situations is not trivial, since scheduling one task in the graph may possibly constrain any other task in the graph.

The algorithm described in Figures 6.9, 6.10, 6.11, 6.12, and 6.13, works by propagating constraint information forward and backward through the application graph  $G$ . The input  $n$  specifies the maximum number of hops allowed for two processors to communicate with each other. In this chapter we will concentrate on single-hop communication, where  $n = 1$ . First an edge-reversed copy  $\hat{G}$  of the application graph  $G$  is

**Algorithm 6.3:** BFSFORWARD( $G, s, n, \text{endNodes}, \text{bottomNodes}, F$ )

**input:** application graph  $G$

**input:** node  $s$  in  $G$  being considered

**input:** set of discovered **endNodes**

**input:** maximum hop communication allowed  $n$

**output:** stack of newly discovered **bottomNodes**

**input/output:** array  $F$  of sets of feasible processors for each node in  $G$

**local variables:** queue of nodes  $Q$ , array **dist** of distances for each node

**local variables:** set  $R$  of processor numbers, application graph nodes  $w, v$

**for all**  $w \in G$

**do**  $\{ \text{dist}[w] = -1 \quad \text{dist}[s] = 0$

$Q \leftarrow \{s\}$

**while**  $(Q \neq \emptyset)$

$v = \text{head}(Q)$

**for all**  $w \in \text{Adj}[v]$

**if**  $\text{dist}[w] < 0$

**do**  $\left\{ \begin{array}{l} \text{ENQUEUE}(Q, w) \\ \text{dist}[w] = \text{dist}[v] + 1 \\ F[w] = F[w] \cap Rf^n(F[v]) \\ \text{if } \text{outdegree}(w) = 0 \\ \text{then } \{ \text{PUSH}(w, \text{bottomNodes}) \} \end{array} \right.$

Figure 6.11: Function bfsForward().

**Algorithm 6.4:** BFSBACKWARD( $\hat{G}, H, s, n, \text{endNodes}, \text{topNodes}, F$ )

**input:** edge-reversed application graph  $\hat{G}$

**comment:** for every node  $u \in \hat{G}$ ,  $v = \hat{G}[u]$  gives corresponding node in  $G$

**input:** array of nodes  $H$

**comment:** for any node  $v \in G$ ,  $u = H[v]$  references corresponding node in  $\hat{G}$

**input:** node  $s$  in  $G$  being considered

**input:** set of discovered **endNodes**

**input:** maximum hop communication  $n$  allowed

**output:** stack of newly discovered **topNodes**

**input/output:** array  $F$  of sets of feasible processors for each node in  $G$

**local variables:** queue of nodes  $Q$ , array **dist** of distances for each node

**local variables:** set  $R$  of processor numbers, application graph nodes  $w, v, \hat{s}$

**local variables:** nodes in  $\hat{G}$ :  $\hat{w}, \hat{v}$

**for all**  $w \in \hat{G}$

**do**  $\{\text{dist}[w] = -1 \quad \hat{s} = H[s]$

$\text{dist}[\hat{s}] = 0$

$Q \leftarrow \{\hat{s}\}$

**while**  $(Q \neq \emptyset)$

$v = \text{head}(Q)$

**for all**  $w \in \text{Adj}[v]$

**if**  $\text{dist}[w] < 0$

$\text{ENQUEUE}(Q, w)$

$\hat{w} = H[w]$

$\hat{v} = H[v]$

$\text{dist}[w] = \text{dist}[v] + 1$

$F[\hat{w}] = F[\hat{w}] \cap Rb^n(F[\hat{v}])$

**if**  $\text{outdegree}(w) = 0$

**then**  $\{\text{PUSH}(\hat{w}, \text{topNodes})$

**do**

**do**

**then**

**then**

Figure 6.12: Function bfsBackward().

**Algorithm 6.5:** FEASIBLE( $G, \hat{G}, H, s, p, n, F, \text{commit}$ )

**input:** application graph  $G$

**input:** edge-reversed application graph  $\hat{G}$

**comment:** for every node  $u \in \hat{G}$ ,  $v = \hat{G}[u]$  gives corresponding node in  $G$

**input:** array of nodes  $H$

**comment:** for any node  $v \in G$ ,  $u = H[v]$  references corresponding node in  $\hat{G}$

**input:** node  $s$  in  $G$  being considered

**input:** processor  $p$  being considered

**input:** maximum hop communication  $n$  allowed

**input:** boolean value **commit** determines if changes to  $F$  are saved

**input/output:** array  $F$  of sets of feasible processors for each node in  $G$

**local variables:** local copy of  $F$  **Flocal**, set of discovered **endNodes**

**local variables:** stack of nodes **topNodes**, application graph nodes  $v$ ,  $v_f$ ,  $b_f$

**local variables:** sets of processor numbers  $f$  and  $r$

**if**  $p \notin F[s]$

**then** {*return* -1

Flocal =  $F$

Flocal[ $s$ ] = { $p$ }

PUSH(topNodes,  $s$ )

**while** topNodes  $\neq \emptyset$

**while** topNodes  $\neq \emptyset$

        POP(topNodes,  $v_f$ )

**do** { INSERT(endNodes,  $v_f$ )

            BFSFORWARD( $G, s, n$ , endNodes, bottomNodes, Flocal)

**while** bottomNodes  $\neq \emptyset$

            POP(bottomNodes,  $v_b$ )

**do** { INSERT(endNodes,  $v_b$ )

                BFSBACKWARD( $\hat{G}, H, s, n$ , endNodes, topNodes, Flocal)

**if** commit

**then** {  $F = \text{Flocal}$

flexibility = 0

**for all**  $v \in G$

**do** { flexibility = flexibility + size(Flocal)

**return** (flexibility)

Figure 6.13: Function feasible() determines feasibility and flexibility (degree of constraint) for scheduling task  $s$  on processor  $p$ .

created. When making a scheduling move (introducing a new constraint at a task  $s$ ), the constraint information is propagated forward using breadth first search from  $s$  through  $G$ . When an *endnode* (task with no successors) is discovered during the forward phase for the first time, it is added to a stack (named **endNodes**).

At the end of the forward phase, the backward phase begins. Each endnode is removed from the stack and the constraint information is propagated backward by performing breadth first search from the endnodes through  $H$ . While propagating backward, newly discovered endnodes of  $H$  are added to a second stack. These endnodes are removed from the stack, and search continues in the forward direction. The process continues until there are no newly found endnodes.

We define  $Rf^1(S)$  and  $Rb^1(S)$  for sets of processors reachable from  $S$  in one hop. Then for multiple hops

$$Rf^2(S) = Rf^1(Rf^1(S))$$

$$Rb^2(S) = Rb^1(Rb^1(S))$$

$$Rf^n(S) = Rf^1(Rf^{n-1}(S))$$

$$Rb^n(S) = Rb^1(Rb^{n-1}(S))$$

We define the functions **bfsForward()** and **bfsBackward()** which use breadth first search to propagate constraint information for a task  $s$  in a graph  $G$  in the forward and backward direction.

The **feasible()** function described in Figure 6.13 returns an integer equal to the sum of the sizes of the constraint sets for all nodes in the application graph  $G$  when scheduling a task  $G$  on a processor  $p$ , given an input  $n$  corresponding to the maximum number of communication hops allowed. If  $s$  is not feasible on  $p$ , the function returns  $-1$ .

(scheduling step) at specified point in feasible	feasible sets
(1) schedule A on 2: end of feasible()	A:[2] B:[1,2,3] C:[0,1,2,3] D:[1,2,3] E:[0,1,2,3] F:[0,1,2,3]
(2) schedule B on 1: after bfsForward() from B	A:[2] B:[1] C:[0,1,2,3] D:[1,2,3] E:[0,1,2,3] F:[1]
(2) schedule B on 1: after bfsBackward() from F	A:[2] B:[1] C:[0,1,2,3] D:[1,2] E:[0,1,2,3] F:[1]
(2) schedule B on 1: after bfsForward() from A	A:[2] B:[1] C:[0,1,2,3] D:[1,2] E:[1,2,3] F:[1]
(2) schedule B on 1: after bfsBackward() from E	A:[2] B:[1] C:[1,2,3] D:[1,2] E:[1,2,3] F:[1]
(3) schedule C on 3: end of feasible()	A:[2] B:[1] C:[3] D:[2] E:[3] F:[1]
(3 alternate) sched C on 1: end of feasible()	A:[2] B:[1] C:[1] D:[1,2] E:[1] F:[1]

Table 6.1: Feasible sets at various points during scheduling.



Table 6.1 lists the constraint sets for the tasks in the example of Figure 6.8 at various stages of the **feasible** function. We are scheduling  $A$  on 2,  $B$  on 1, and  $C$  on either 3 or 1. Here we can see that after scheduling task  $B$  on processor 1, processor 0 is not a feasible choice for task  $C$ . In Table 6.1, the last row corresponds to the second choice of scheduling  $C$  on processor 1.

## 6.4 Complexity of the Constraint Algorithm

The **bfsForward** function will be called once for the task  $s$  being considered, and once for each task in  $G$  with no predecessors (endnode in  $\hat{G}$ ). The **bfsBackward** function will be called once for each task in  $G$  with no successors (endnode in  $G$ ). The complexity of breadth first search is  $O(v + e)$  for a graph with  $v$  nodes and  $e$  edges. The **bfsForward** and **bfsBackward** functions require a set intersection of two sets of size  $O(N)$  where  $N$  is the number of processors in the system. This has complexity  $O(N \log N)$ . Functions **Rb<sup>n</sup>** and **Rf<sup>n</sup>** also have complexity  $O(N \log N)$ . The overall complexity is therefore

$$O(v(v + e)N \log N) \quad (6.1)$$

This is a reasonable complexity figure in the embedded systems domain, where compile/synthesis time tolerance is significantly higher compared to general-purpose computation (e.g., see [74]).

For interconnection graphs that are strongly connected, such as those in which all links are bidirectional, **Rf<sup>n</sup>**( $S$ ) =  $\Phi$  (the set of all processors) and **Rb<sup>n</sup>**( $S$ ) =  $\Phi$  after some number of hops  $h \leq N$ , and the breadth first searches do not need to proceed for distances further than  $h$  where  $h$  is the maximum hop constraint given beforehand. In this case the complexity is  $O(vhN \log N)$ .

## 6.5 Incorporating Feasibility and Flexibility into Scheduling

The general class of list scheduling algorithms can easily be adapted to produce single-hop (or  $n$ -hop) schedules by incorporating our constraint algorithm. This is advantageous because it allows us to leverage a large library of useful scheduling techniques.

In list scheduling, a priority list  $L$  of tasks is constructed. The priority list  $L$  is a linear ordering  $(\nu_1, \nu_2, \dots, \nu_{|V|})$  of the tasks in the application graph  $G = (V, E)$  such that for any pair of distinct tasks  $\nu_i$  and  $\nu_j$ ,  $\nu_i$  is to be given higher scheduling priority than  $\nu_j$  if and only if  $i < j$ . Each task is mapped to an available processor as soon as it becomes the highest-priority task according to  $L$  among all tasks that are ready. This process is repeated until all tasks are scheduled.

The concepts of feasibility and flexibility, which were developed in Section 6.3, can be incorporated into the general framework of list scheduling by restricting the set of candidate processors to include only those that are feasible at the given scheduling step, and by taking flexibility into account in designing the priority metric through which tasks are ordered.

In the context of single-hop communication across arbitrary interconnection patterns, the incorporation of feasibility considerations is required (to avoid scheduler deadlock, as discussed in Section 6.3), while incorporation of flexibility is optional. Furthermore, there are many possible ways to consider flexibility in the task prioritization process. We show in Section 6.6 that even simple techniques for incorporating flexibility information can lead to large performance improvements for a targeted class of architectures.

## 6.6 Scheduling Experiments using Flexibility

As mentioned earlier, our scheduling technique operates in conjunction with a given list scheduling strategy. In our experiments, we employed the DLS algorithm [96] as the underlying list scheduling strategy, although, as explained in Section 6.3, any list scheduling algorithm could have been used.

We examined a set of DSP application benchmarks and scheduled them using two different scheduling modes, one that incorporates only feasibility information (to avoid deadlock), and another that takes both feasibility and flexibility into account. We refer to these as the *feasibility-only* and *feasibility-flexibility* modes, respectively. To evaluate the performance across a range of connectivity levels, we scheduled the applications onto networks with varying degrees of connectivity.

In the feasibility-only mode, the processor  $P$  considered for a given task  $\nu$  at each scheduling step was restricted to be in the feasible set  $\Psi[\nu]$  for  $\nu$ , as described in Section 6.3, and no modification was made to the task prioritization metric of the underlying list scheduling strategy (DLS).

In the feasibility-flexibility mode, the processor  $P$  considered at each scheduling step was again restricted to be in the feasible set for  $\nu$ ; however, whenever two processor assignments for  $\nu$  resulted in equal priority levels  $L(\nu)$ , where  $L$  represents the priority metric of the original DLS algorithm, priority was given to the assignment that resulted in a higher value of flexibility. In other words, priority was given to assignments that offered greater flexibility for future scheduling decisions.

For each application, we chose a number  $N$  of processors, then generated a fully connected network with  $N(N - 1)$  links. We scheduled the application using both feasibility-only and feasibility-flexibility modes onto this network. Next we removed one link from the network at random, and again scheduled the application using both

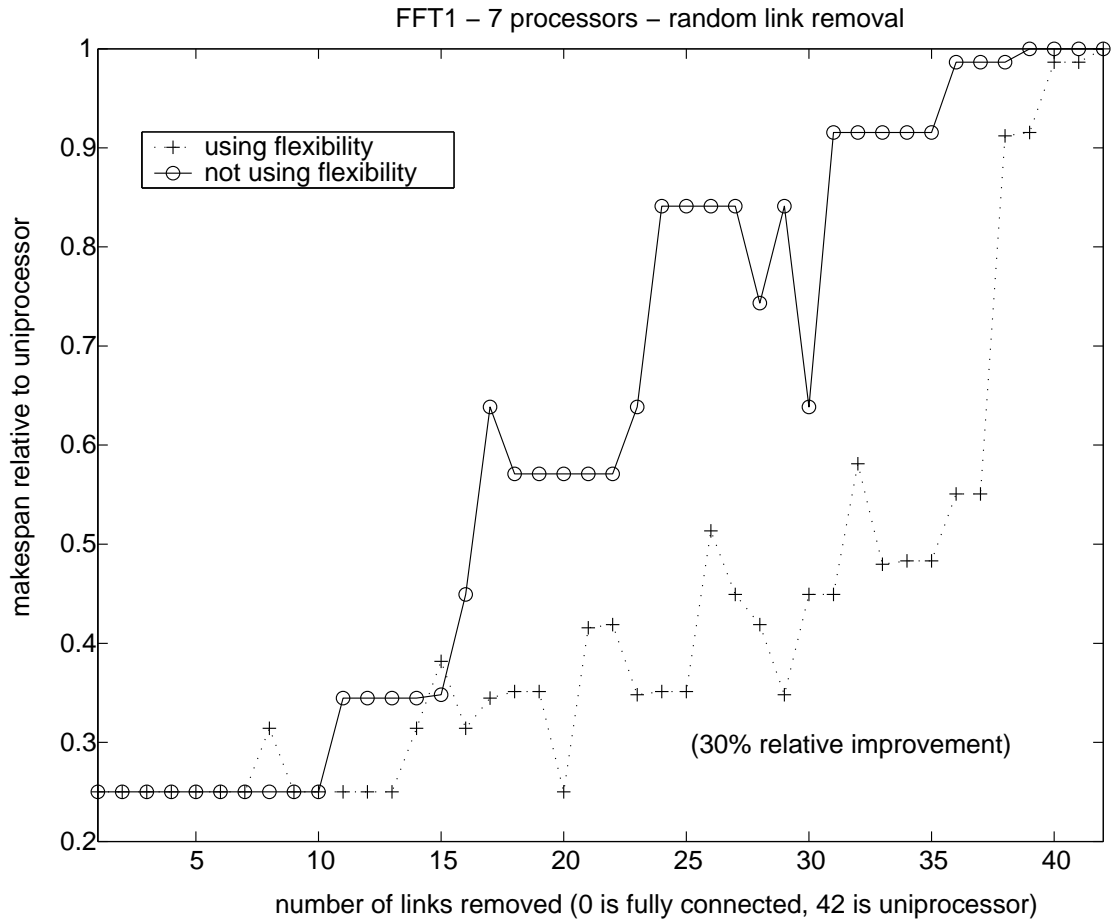


Figure 6.14: Makespans for schedules constructed using DLS plus flexibility computation with and without considering the processor flexibility metric.

scheduling modes. We continued this process of removing links until no links remained, resulting with all the tasks scheduled on a single processor. We define the relative improvement of the feasibility-flexibility mode over the feasibility-only mode by comparing the average makespan over all link configurations.

The result of this experiment for an FFT application is shown in Figure 6.14. If we compare the average makespan for the schedules generated by feasibility-flexibility mode (the top curve in Figure 6.14) with the average makespan of the schedules gen-

Application	N	Improvement(%)
FFT1	7	30
Karp10	6	26
Irr	8	17
Qmf4	7	19
NN16-3-4	8	21
Sum1	6	8
Laplace	7	23
FFT2	7	15

Table 6.2: Relative makespan improvement obtained by using flexibility information in the scheduling process.

erated without incorporating flexibility (the bottom curve in Figure 6.14) we see a 30% relative improvement when the scheduling algorithm incorporates the flexibility metric.

Table 6.2 summarizes this relative improvement for several other DSP applications. We performed experiments with the following application graphs: FFT1, Irr, FFT3, Karp10, Qmf4, Laplace, Sum1, and NN16-3-4. The FFT graphs are different implementations of the fast Fourier transform from Kahn [56] and contain 28 nodes each. Karp10 refers to the Karplus-Strong music synthesis algorithm with 10 voices (21 nodes), and Qmf4 is a quadrature mirror filter bank with 14 nodes. Laplace is a Laplace transform, Irr is an adaptation of a physics algorithm, and sum1 is an upside down binary tree representing the sum of products computation. A neural network classifier algorithm with 16 input nodes, 3 intermediate layers, and 4 output nodes labeled NN6-3-4 was also tested.

Application	N	Reduction in comm. energy(%)	Makespan increase for single hop
FFT1	7	16	8
Karp10	6	24	4
Irr	8	16	(2)
Qmf4	7	32	3
NN16-3-4	8	58	2
Sum1	6	1	4
Laplace	7	4	(3)
FFT2	7	12	2

Table 6.3: Reduction in communication of single hop schedule over three-hop schedule.

### 6.6.1 Power Reduction with Single Hop Communication

As mentioned earlier, it is advantageous to limit interprocessor communication to a low number of hops because the energy required is proportional to the number of electrical-to-optical conversions. In order to quantify this effect, we scheduled the benchmark applications using our modified scheduling technique, which takes the number of hops as an input parameter. We scheduled the benchmarks with hop constraints of one hop and three hops, and compared the communication energy required. For our purposes, we assumed all communication tasks transferred the same number of bits, so the energy cost of all IPC actors was equal. With a three-hop limit, the scheduler is free to choose any communication path that involves three or fewer hops and is thus less constrained in its scheduling choices than with a one-hop limit. Table 6.3 shows the reduction in the required communication energy for single-hop schedules over three-hop schedules for the benchmark applications. We would expect that in general the schedules con-

structed with the three-hop limit would have a lower makespan, since the scheduler is less constrained—the set of moves available to the scheduler at any point using three hops is a superset of the moves available when limited to one hop. For these benchmarks, however, we found that any undesirable effect of the additional constraint for single-hop schedules was very small, as can be seen in Table 6.3. In two of the benchmarks (Irr and Laplace), the makespan was in fact better (lower) when we limited the scheduler to single hops.

## **6.7 Summary of Flexibility Work**

Optical interconnect technology is promising for global communication in embedded multiprocessors, since the interconnection patterns can flexibly be streamlined and re-configured to match the target applications. However, due to the power consumption characteristics of optical links, it is useful to restrict communication across them to low-hop transfers. We have demonstrated an effective algorithm for determining the set of feasible processors that will avoid schedule deadlock in a single-hop schedule, and a useful metric, called communication flexibility, for the degree to which a given scheduling decision constrains future decisions (in the context of the given communication topology). We used this algorithm and the flexibility metric in conjunction with the DLS algorithm to map several DSP applications across a wide range of interconnect topologies. The results depicted in Figure 6.14 and 6.2 demonstrate both the soundness of our feasibility computation techniques, and the utility of our flexibility metric in guiding the scheduling process.

## Chapter 7

# Synthesizing an Efficient Interconnect Network

Embedded systems typically run a limited and fixed set of applications. We can use this application-specific information to optimize the interconnection network. For our purposes, an optimal network is defined in the context of a set of applications and constraints. The constraints may include the latency, throughput, and power consumption for the given applications, along with cost and area constraints of the overall system.

This problem is important for today's system-on-chip (SoC) designs utilizing electronic interconnects as well as future designs that might utilize optical interconnects. SoC design is moving toward a paradigm where reusable components called IP (for intellectual property) from different vendors can be combined to rapidly create a design. IP interface standards are being developed that define the services one IP component (or *IP block*) is capable of delivering, and that enable IP blocks to work with on-chip buses and other interconnection networks. The SoC designer's task is then to choose the appropriate IP blocks, map the application tasks onto these blocks, and to construct a communication network and corresponding glue logic to connect these IP blocks. As transistor density increases, more IP blocks can be placed on a single chip, and the number of possible interconnections (links) between them increases. The longest wires on



the chip are usually due to these links. These wires contribute to delay and limit the maximum achievable clock rate. Also, routing these interconnections is a significant challenge for the EDA tools. Therefore, if we can minimize the number of links required in the high level design stage, placement and routing can be improved in the back end of the design process and performance will increase.

In a system utilizing optical interconnects, cost and area constraints dictate the total number of transmitters and receivers in the system (i.e., total number of optical links). Routing constraints from local partitions to their associated VCSEL transmitters and detectors dictate a maximum fanout for each local partition. An optimum interconnect is then one that minimizes the number of links while enabling the application to meet the power, latency, and throughput constraints.

Realistic optical networks may incorporate relatively high, but not necessarily complete (fully connected), levels of connectivity. Even in fully-connected systems, such as FAST-Net [48], it is still desirable from the viewpoint of power and heat dissipation to have a minimal interconnect mapping, since for a given application, non-essential transmitters can be turned off. In other optical processing implementations, the interconnect network can be reconfigured between applications [54].

The freedom to optimize interconnection patterns opens up a vast design space, and thus the design of an optimal interconnect structure for a given application or set of applications is a significant challenge. In this chapter, we illustrate both probabilistic and deterministic interconnection synthesis algorithms. A key distinguishing feature to our interconnect synthesis algorithms is that they work in conjunction with a scheduling strategy—most existing interconnect synthesis algorithms assume a given schedule.

## 7.1 Greedy Interconnect Synthesis Algorithm

We developed a greedy, heuristic algorithm, which we call the *two-phase link adjustment* (TPLA) algorithm [6], to synthesize an interconnect and an associated multiprocessor schedule for a given application. The TPLA algorithm starts with a fully connected network, and operates in *down* and *up* phases. Input to the algorithm is either a makespan constraint for the application, or a constraint on the total number of links.

Each step of the down phase in TPLA removes one link, while each step of the up phase adds one link. One step of the down phase consists of assigning each existing link a score based on the schedule makespan resulting from its removal, and removing the link with the lowest score. A history of scores is kept for each link. For the first pass through the down phase, ties between links are broken randomly. On subsequent phases, the link history is used to break ties. The down phase continues until all the links are removed.

Conversely, one step of the up phase in TPLA consists of assigning a score to each missing link based on the makespan resulting from its addition. The up phase continues until the network is fully connected. Repeated, alternating invocations of down and up phases are executed for some time limit (determined by the user), and the best result found is taken. Given a makespan constraint, this best result minimizes the number of links. Alternatively, given a constraint on the number of links, the best result minimizes the makespan.

### 7.1.1 Experiments with TPLA

We evaluated the TPLA algorithm on a neural network classifier application called RBFNN, consisting of 16 input nodes, 3 intermediate layers, and 4 output nodes. This

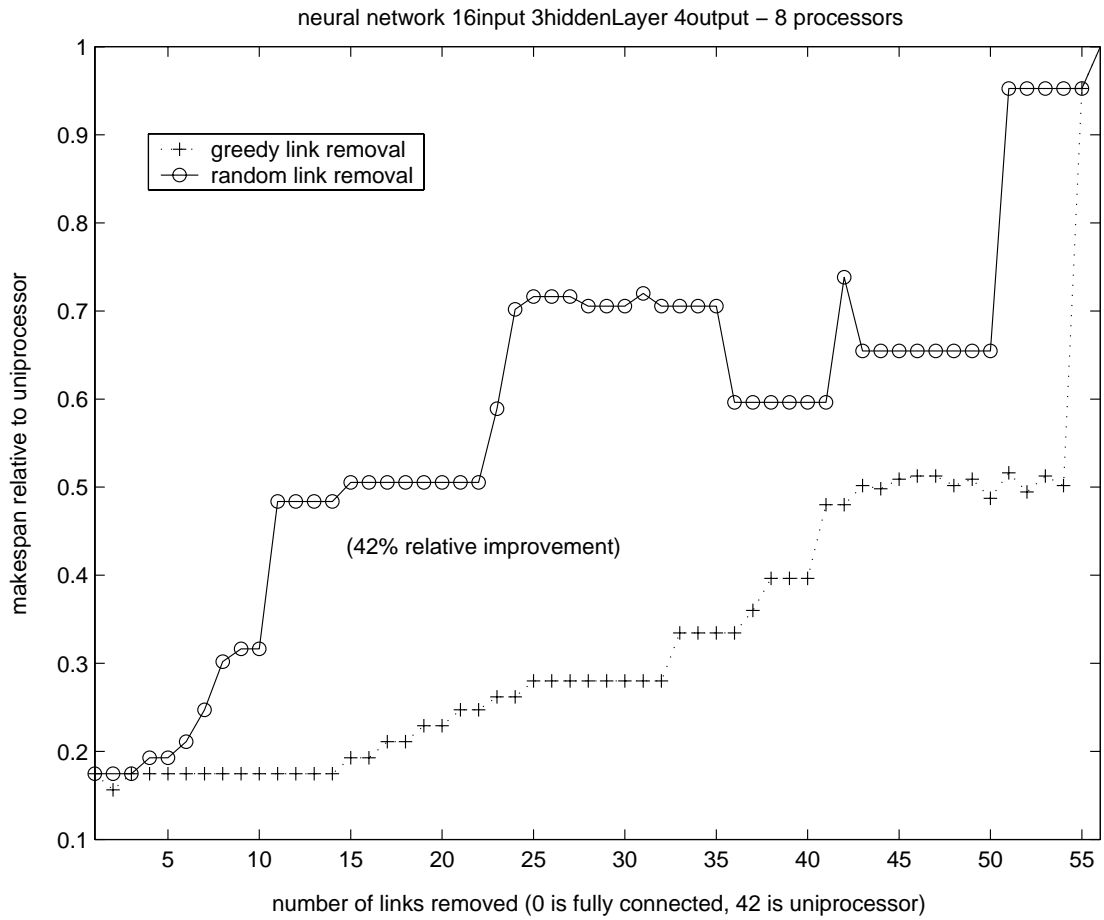


Figure 7.1: Link synthesis using the TPLA algorithm.

benchmark was chosen in part since it exhibits a large amount of inter-processor communication. The scheduling algorithm used was the DLS algorithm [96] modified to incorporate the flexibility metric, as detailed in Section 6.5. The bottom curve of Figure 7.1 shows the best makespan achieved for each level of connectivity between 0 and fully connected, after one down phase and one up phase. This gives a Pareto curve of the trade-off between number of links and makespan for the application. For purposes of comparison, the upper curve of Figure 7.1 shows the makespan achieved by starting with fully connected and randomly removing one link at a time. The TPLA algorithm

shows a significant improvement (42% relative improvement) over random removal, and is thus a promising starting point for developing more sophisticated link synthesis algorithms. More broadly, it demonstrates the effectiveness of joint scheduling and interconnect synthesis.

## 7.2 Link Synthesis using Genetic Algorithm

We developed a genetic algorithm (GA) based interconnect synthesis algorithm. This algorithm also employs the dynamic level scheduling (DLS) algorithm [5] modified for arbitrary interconnection networks as the underlying list scheduling strategy, although any list scheduling algorithm could have been used. The algorithm takes into account constraints on the total number of links  $l_{\max}$  and a maximum fanout for each processor  $f_{\max}$ , as described earlier and motivated by area and cost constraints for the system.

### 7.2.1 Genetic Algorithm Overview

Genetic algorithms will be described in more detail in Chapter 8—we give a brief overview here in order to explain the link synthesis algorithm. When a genetic algorithm is used to solve an optimization problem, it is necessary to be able to represent a single solution to the problem with a single data structure. This representation is often called a *chromosome* or an *individual*. The quality or *fitness* of a given solution is evaluated using an *objective function*. Genetic algorithms are capable of both broad search (exploration) and local search (exploitation) of a search space. They are often preferred than gradient search methods because they avoid local minima, and do not require a smooth search space.

The basic steps of a genetic algorithm are shown in Figure 7.2. The genetic algo-

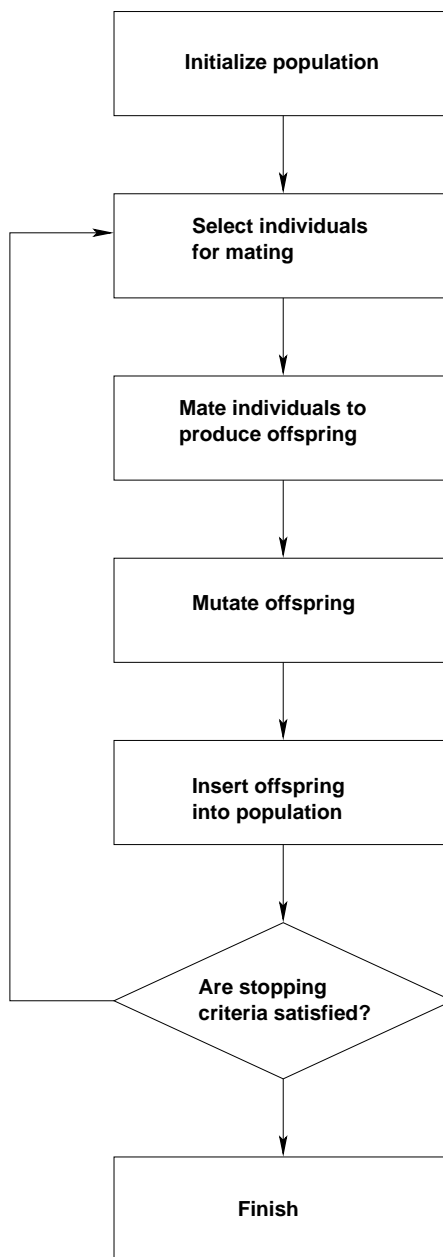


Figure 7.2: Basic steps of a GA.

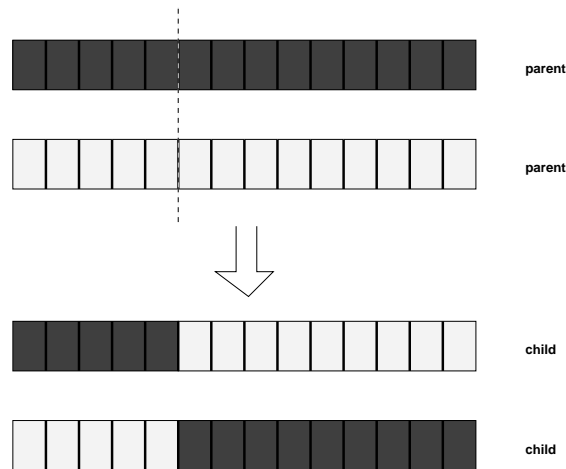


Figure 7.3: Crossover operator applied to array chromosome.

rithm creates an initial *population* of candidate solutions using an initialization operator. Often the initial population is distributed randomly over the search space. The genetic algorithm first selects individuals from the population and performs *crossover* and *mutation* operations on these individuals. Traditional crossover generates two *children* from two *parents* in a population. This is depicted in Figure 7.3 for a chromosome whose representative data structure is an array. A *crossover point* is chosen, shown by the dashed vertical line in Figure 7.3, and the child chromosome is formed by the elements from the first parent chromosome to the left of the crossover point and the elements from the second parent to the right of the crossover point. The mutation operator specifies a procedure for changing (mutating) an individual. The specifics of the mutation depend on the data structure used to represent an individual. A typical mutation operator for an individual represented by a binary string flips the bits in the string with a given probability (the *mutation probability*). One *generation* of a genetic algorithm consists of performing crossover and mutation on individuals in the population. There are many possibilities for evolving the population. A *simple* GA uses non-overlapping popula-

tions. Each generation creates an entirely new population of individuals. A *steady state* GA uses overlapping populations, in which a fraction of the population is replaced in each generation. In an *incremental* GA each generation consists of only one or two children.

## 7.2.2 Problem representation

In our algorithm, the individuals are bit vectors corresponding to a given interconnect topology. The fitness function for a chromosome in our interconnect synthesis algorithm is described by

$$\text{fitness} = M(1 + P_f + P_l) \quad (7.1)$$

where  $M$  is the makespan (latency) calculated by the modified DLS algorithm for the interconnect topology of the chromosome,  $P_f$  (equation 7.6) is a penalty based on violating the fanout constraint  $f_{\max}$ , and  $P_l$  (equation 7.7) is a penalty based on violating the maximum link constraint  $l_{\max}$ .

We define a *link vector* as a bit vector with one entry for each possible interconnection between two processors. For a system with  $N$  processors, there are  $N(N - 1)$  entries in the link vector. The link vector for a four processor system would be denoted as

$$\vec{l} = (l_{01}l_{02}l_{03}l_{10}l_{12}l_{13}l_{20}l_{21}l_{23}l_{30}l_{31}l_{32}) \quad (7.2)$$

where  $l_{ij}$  equals one if there is a connection from processor  $i$  to processor  $j$  and zero otherwise. We define  $l_{ij} \equiv 0$  if  $i = j$ . We also write  $\vec{l}$  as

$$\vec{l} = (\vec{l}_0\vec{l}_1 \dots \vec{l}_{N-1}) \quad (7.3)$$

where  $\vec{l}_k$  describes the (outgoing) connections for processor  $k$ . We will refer to the  $\vec{l}_k$  as

*processor link vectors*. We define the fanout of processor  $i$  by

$$f_i = \sum_{j=0}^{N-1} l_{ij} \doteq \|\vec{l}_i\| \quad (7.4)$$

Then the number of links is given as

$$n_l = \sum_{i=0}^{N-1} f_i \quad (7.5)$$

while the fanout penalty is given by

$$P_f = \sum_{i=0}^{N-1} P_i \quad (7.6)$$

where  $P_i = \max(0, (f_i - f_{\max}))$ . The link penalty is given by

$$P_l = \max(0, (n_l - l_{\max})). \quad (7.7)$$

### 7.2.3 Fanout Constraints

In a real system, cost and area constraints will place a limit on the processor fanout. For example, in a free-space optical system such as FAST-Net [48], each link requires a dedicated VCSEL/photoreceiver pair. In the WDM-based system proposed in Chapter 5, a separate wavelength is required to transmit to each processor, and each processor requires a tunable source. In this case there is a physical limit on the number of resolvable wavelengths, given by  $n = B/\Gamma$  where  $B$  is the fiber bandwidth and  $\Gamma$  is the channel spacing. Cost constraints may also limit the number of wavelengths allowed for the tunable sources. For today's WDM systems,  $\Gamma = 50\text{GHz}$  while  $B \cong 4000\text{GHz}$  corresponding to the wavelength range from 1530 to 1565nm (C band) in a fiber. This yields 80 channels. In order to achieve such narrow channel spacing, the temperature of the laser transmitter must be carefully controlled. A lower cost variant to WDM, called coarse wavelength division multiplexing (CWDM) is being deployed in metropolitan



networks. The latest standard proposed by the Full Spectrum CWDM Alliance is a channel spacing  $\Gamma = 20\text{nm}$  ( $\cong 3600\text{GHz}$ ) starting from 1271nm up to 1611nm. The wider channel spacing ( $\cong 72$  times greater) allows lower tolerances on the lasers, and allows them to operate without temperature control.

In any case, it is important to have a link synthesis algorithm that can conform to fanout constraints. Our GA is able to incorporate these constraints in a straightforward manner by implementing the initialization, crossover, and mutation operators as described below.

#### 7.2.4 Crossover and Mutation Operators

We first note that if an individual topology is represented as a binary string as in equation 7.2, then the typical crossover operations like array one-point crossover (Figure 7.3) or two-point crossover will not preserve the fanout constraint. This is illustrated in Figure 7.4 where both parents obey a fanout constraint  $f_{\max} = 2$ , but processor 0 of child X has fanout  $f_{0X} = 3$ . This is because the crossover point can be chosen at any point. If we instead choose to represent the topology by the vector representation of Equation 7.3, fanout constraints are preserved in the crossover operation, since the link vectors for individual processors  $\vec{l}_i$  are never altered. The crossover operation only rearranges the relative position of these link vectors. This is illustrated in Figure 7.5.

We also must ensure that the initial population obeys the link constraint. The initialization operator generates random processor link vectors which each satisfy the fanout constraint Equation 7.4.  $N - 1$  of these vectors are then concatenated to form the link vector.

The mutation operator simply chooses a random bit in the link vector, and sets its value to zero. This removes a link if one existed at this point. Since the mutation

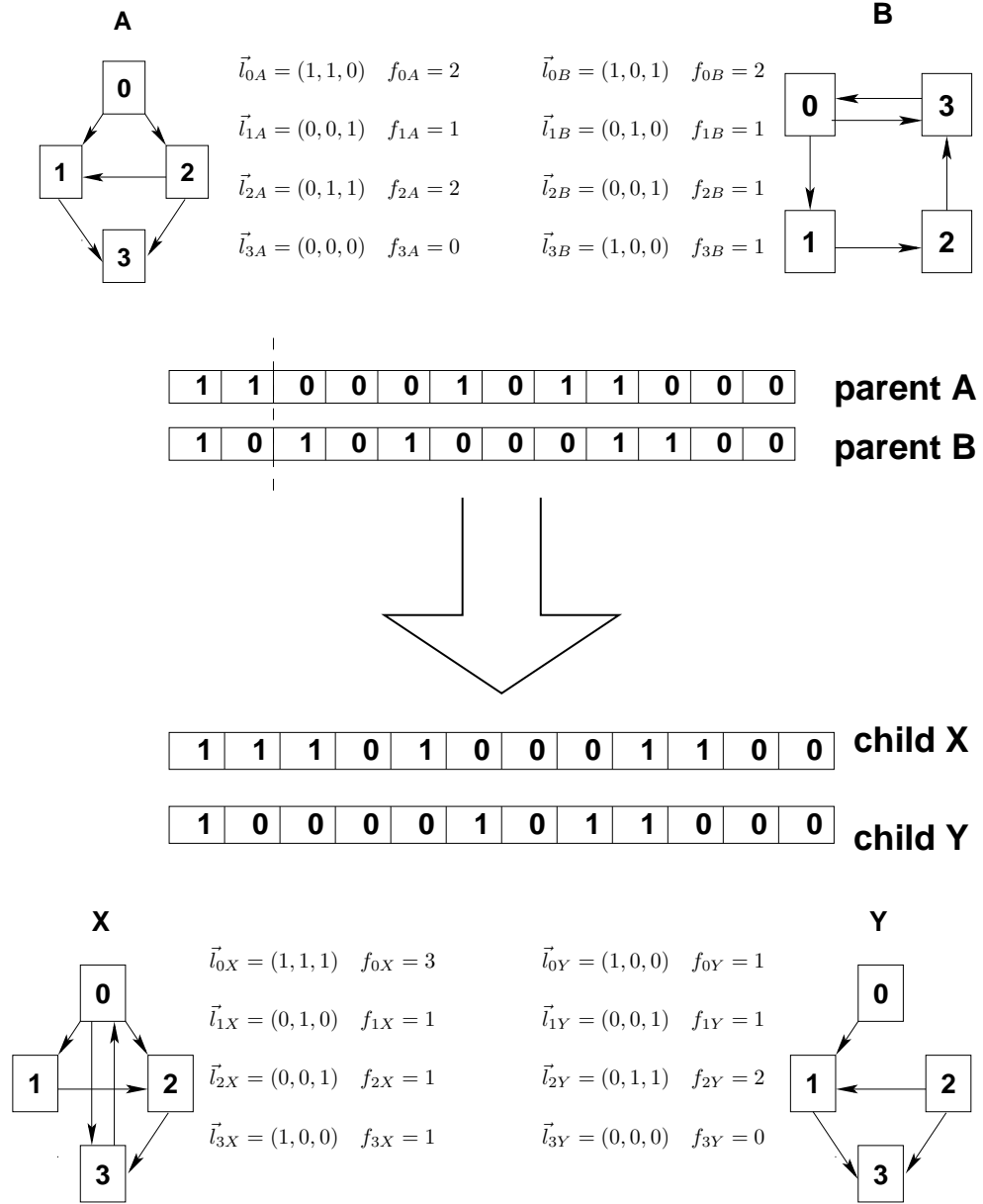


Figure 7.4: Crossover operation for link synthesis using the binary string representation Equation 7.2. Link fanout constraint is not preserved for child  $X$ , where the fanout of processor 0 is  $f_{0X} = 3$ .

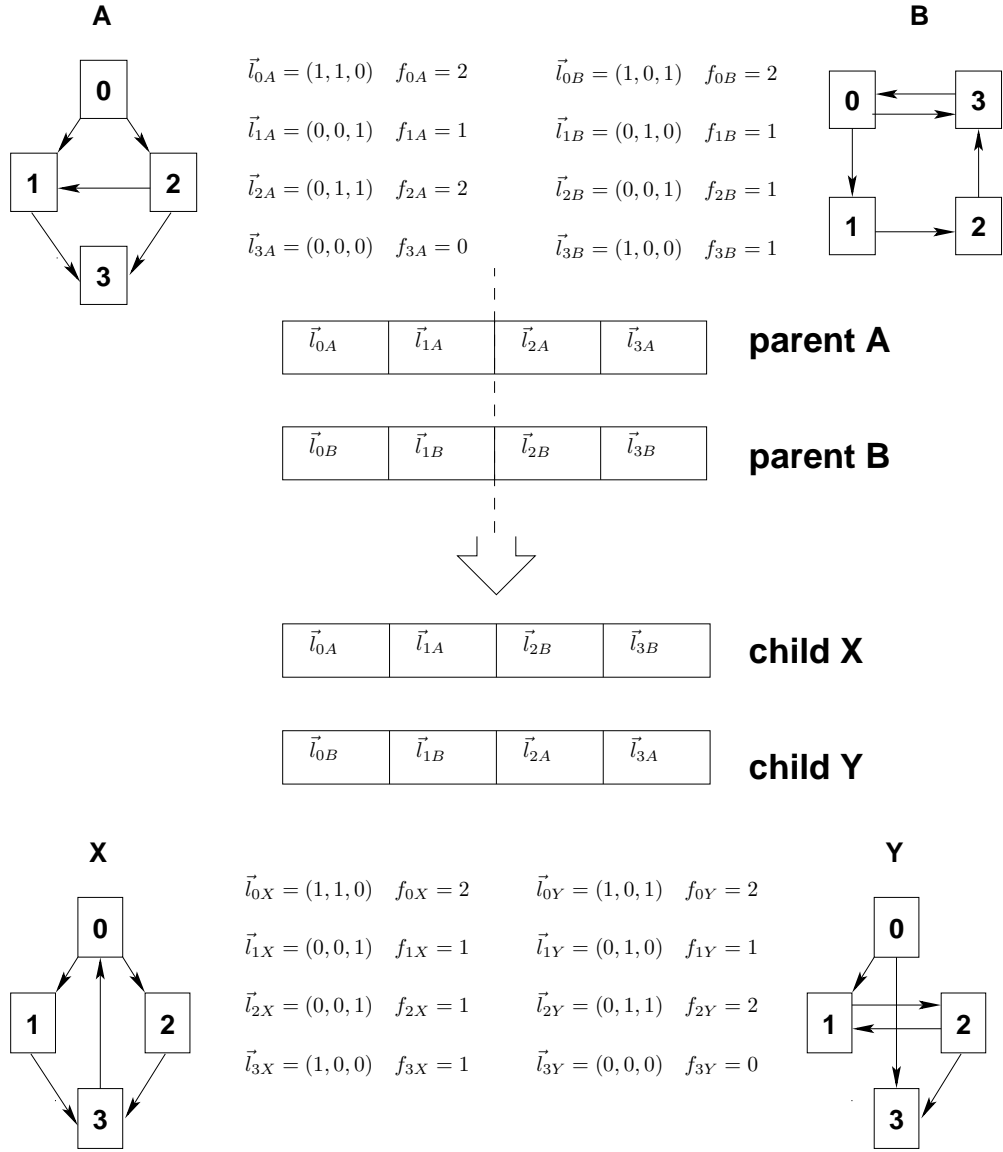


Figure 7.5: Crossover operation for link synthesis using the vector representation Equation 7.3. The fanout constraint  $f_{\max} = 2$  is preserved in the children.

operator only removes links, the fanout constraint is preserved.

### 7.2.5 Experiments

We evaluated our GA-based interconnection synthesis algorithm on the RBFNN application discussed above. We compared the GA-based algorithm to the TPLA algorithm. The genetic algorithm has several advantages over the TPLA algorithm. The first advantage is that it is able to incorporate fanout constraints, which the TPLA algorithm does not. Cost and area considerations often dictate fanout constraints. In a free-space optical system, as already mentioned, fanout is dictated by the number of VCSELs and photoreceivers that can be placed adjacent to a processor. In a WDM system, cost constraints dictate the number of wavelengths used. The second advantage is that, in order to synthesize a network for a given link constraint, the TPLA must evaluate many intermediate topologies that do not meet the link constraint during its construction phases. This makes it much less efficient, especially for systems with a large number of processors. Neither of these algorithms take into account isomorphically unique link topologies, which is the subject of the following section. Figure 7.6 shows the best latency achieved for each level of connectivity between zero connectivity and fully connected for both algorithms. This gives a Pareto curve of the trade-off between number of links and latency for the application. In order to properly compare the different algorithms, the GA run time was limited to the run time required by TPLA. The results show that the algorithm based on the GA performs 21% better (producing lower makespan schedules), when averaged over the different link configurations, for this benchmark.

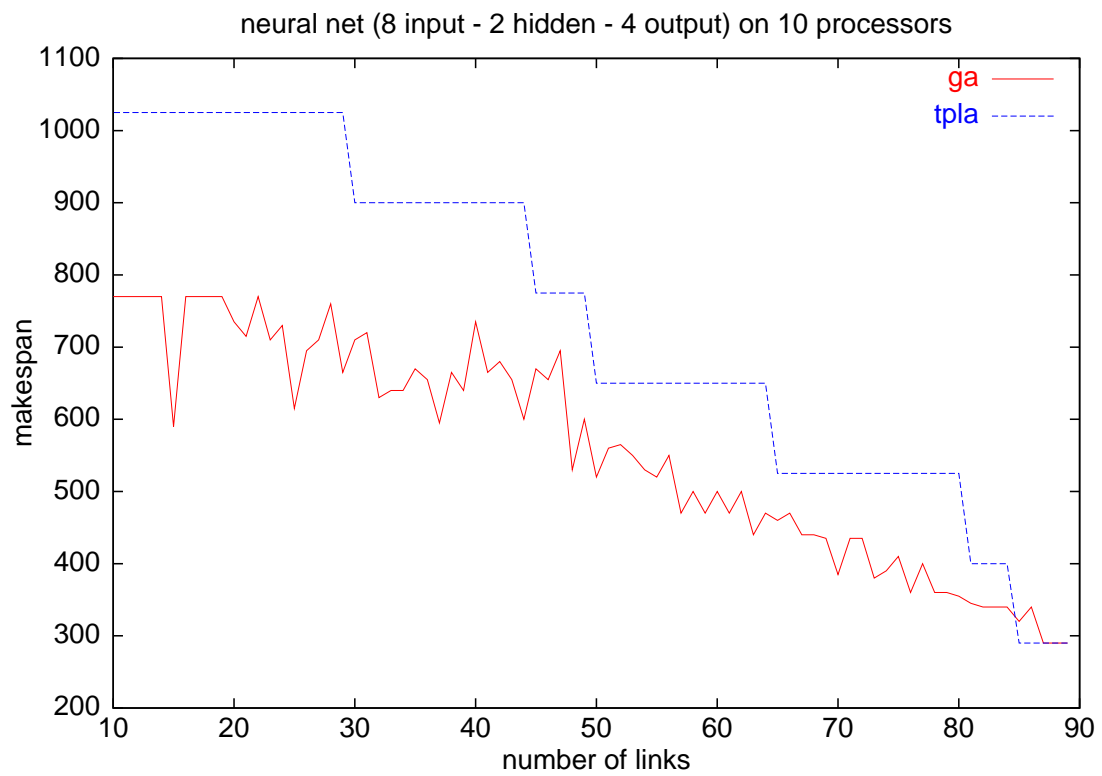


Figure 7.6: Comparison of TPLA and genetic algorithm for neural network application.

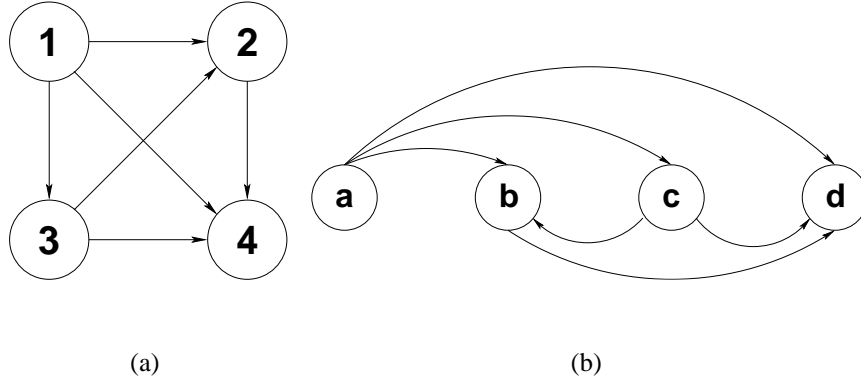


Figure 7.7: Example of two isomorphic graphs.

### 7.3 Using Graph Isomorphism

If we consider systems in which all the processors are identical (homogeneous processor set), then we can pare the design space significantly if we only consider isomorphically unique topology graphs. Two graphs  $G = (V, E)$  and  $G'(V', E')$  are isomorphic if we can relabel the vertices of  $G$  to be vertices of  $G'$ , maintaining the corresponding edges in  $G$  and  $G'$ . For example, the graphs in Figures 7.7(a) and 7.7(b) are isomorphic with the vertices relabelled as follows:  $1 \rightarrow a$ ,  $2 \rightarrow b$ ,  $3 \rightarrow c$ , and  $4 \rightarrow d$ .

Consider a topology graph  $G$  with  $E$  edges and  $N$  nodes where each node corresponds to a processor and each edge corresponds to a link between two processors. The maximum number of edges in  $G$  is  $E_{\max} = N(N - 1)$  corresponding to a fully connected graph (full crossbar interconnect). If all links are bidirectional, the topology graph is undirected and  $E_{\max} = N(N - 1)/2$ . We can represent the graphs with either an adjacency list or adjacency matrix and label each different representation. Then for a graph with  $E$  edges the number of different labellings is given by

$$n_g = \binom{E_{\max}}{E} = \frac{E_{\max}!}{E!(E_{\max} - E)!} = \frac{N(N - 1)!}{E!(N(N - 1) - E)!} \quad (7.8)$$

which increases exponentially with  $N$ . The maximum value of  $n_g$  occurs at  $E = E_{\max}/2$ . However, the number of isomorphically unique graphs  $n_{\text{unique}}$  is much less than  $n_g$ . For very small  $N$ , we can enumerate the different possibilities to show this. Figure 7.8 depicts the different isomorphic graphs for  $N = 4$  processors and  $E = 3$  bidirectional links. There are 20 different graph labellings, but we observe that most are isomorphic—only 3 are isomorphically unique.

For larger  $N$ ,  $n_g$  increases rapidly according to Equation 7.8. We enumerated the possibilities and tested for isomorphism for  $N = 5$  and  $N = 6$  using Brendan McKay's *nauty* program [78], which is currently the fastest published graph isomorphism testing program. The results are shown in Figure 7.9. For  $N = 6$  and  $E = 12$  we observe that there is a 3 order magnitude difference between the  $n_g$  and  $n_{\text{unique}}$ . Also, this ratio increases with  $n_g$ . We would like to exploit this property to pare the design space for link synthesis.

Since  $n_g$  is so large it is impractical to compute and store the isomorphic graphs in advance. Rather, we employ an on-line isomorphic test in order to speed up our deterministic algorithm. This is illustrated in Figure 7.10. In this algorithm we store the topology graphs, and schedule them only if they are not isomorphic with another topology graph previously evaluated. We begin with a connected graph  $G_1$  with  $e = N - 1$  edges, and define a set  $S$  of evaluated graphs. Initially,  $S = G_1$ . We define a parameter  $j_{\max}$  which corresponds to the maximum number of graphs we will consider at a given  $e$ , and a parameter  $G_{\text{best},e}$  which corresponds to the best graph with  $e$  edges.

We construct graph  $G_2$  by adding an edge to  $G_1$ . At this step, there are  $N(N - 1) - e$  possible edges to add. If  $G_2$  is not isomorphic with a graph in  $S$ , we set  $S = S \cup \{G_2\}$  and schedule  $G_2$  using a combination of the DLS scheduling algorithm [97] and the flexibility algorithm as described in Chapter 6. If the throughput using  $G_2$  is higher than

$N = 4$  processors, all links bidirectional

$$E_{\max} = 6$$

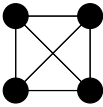
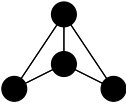

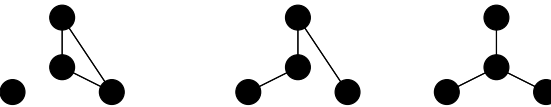

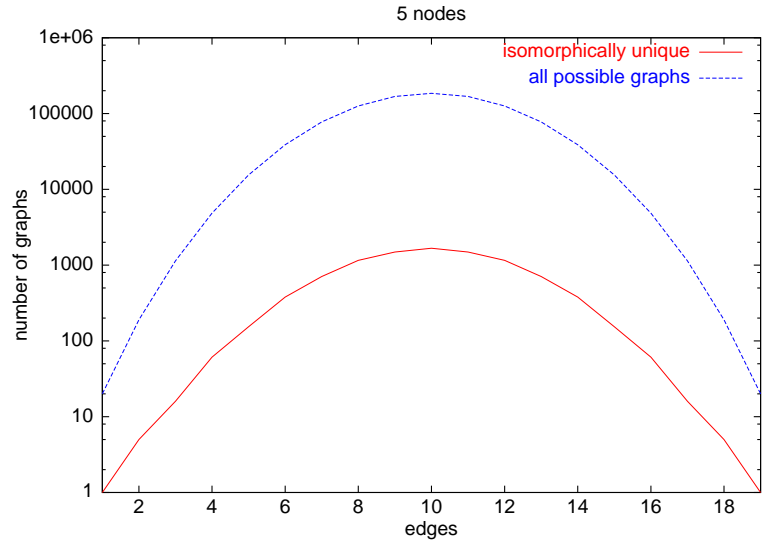
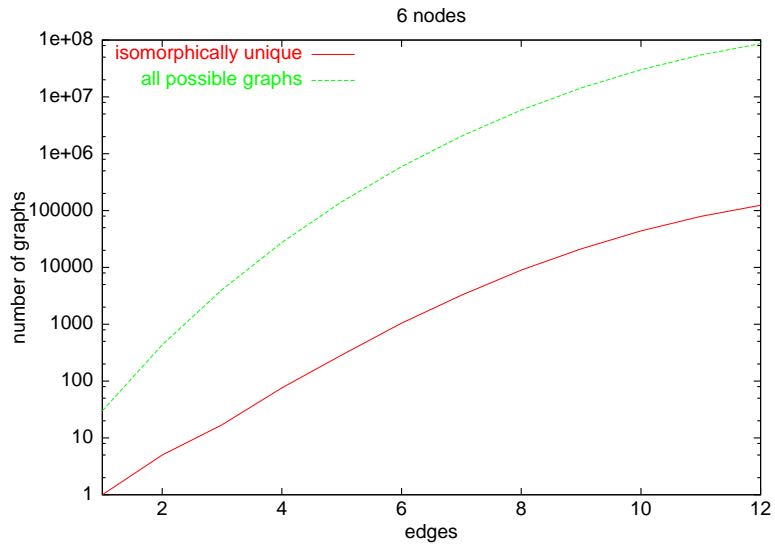
$E$	$n_g$	$n_{\text{unique}}$	
6	1	1	
5	6	1	
4	15	2	
3	20	3	
2	15	2	

Figure 7.8: Isomorphically unique graphs containing  $E$  edges for  $N = 4$  processors. Here we only consider undirected graphs representing bidirectional links in order to make the figure clearer.





(a)  $N = 5$



(b)  $N = 6$

Figure 7.9: A comparison of the number of possible graph labellings  $n_g$  given by Equation 7.8 with the number of these graphs that are isomorphically unique.

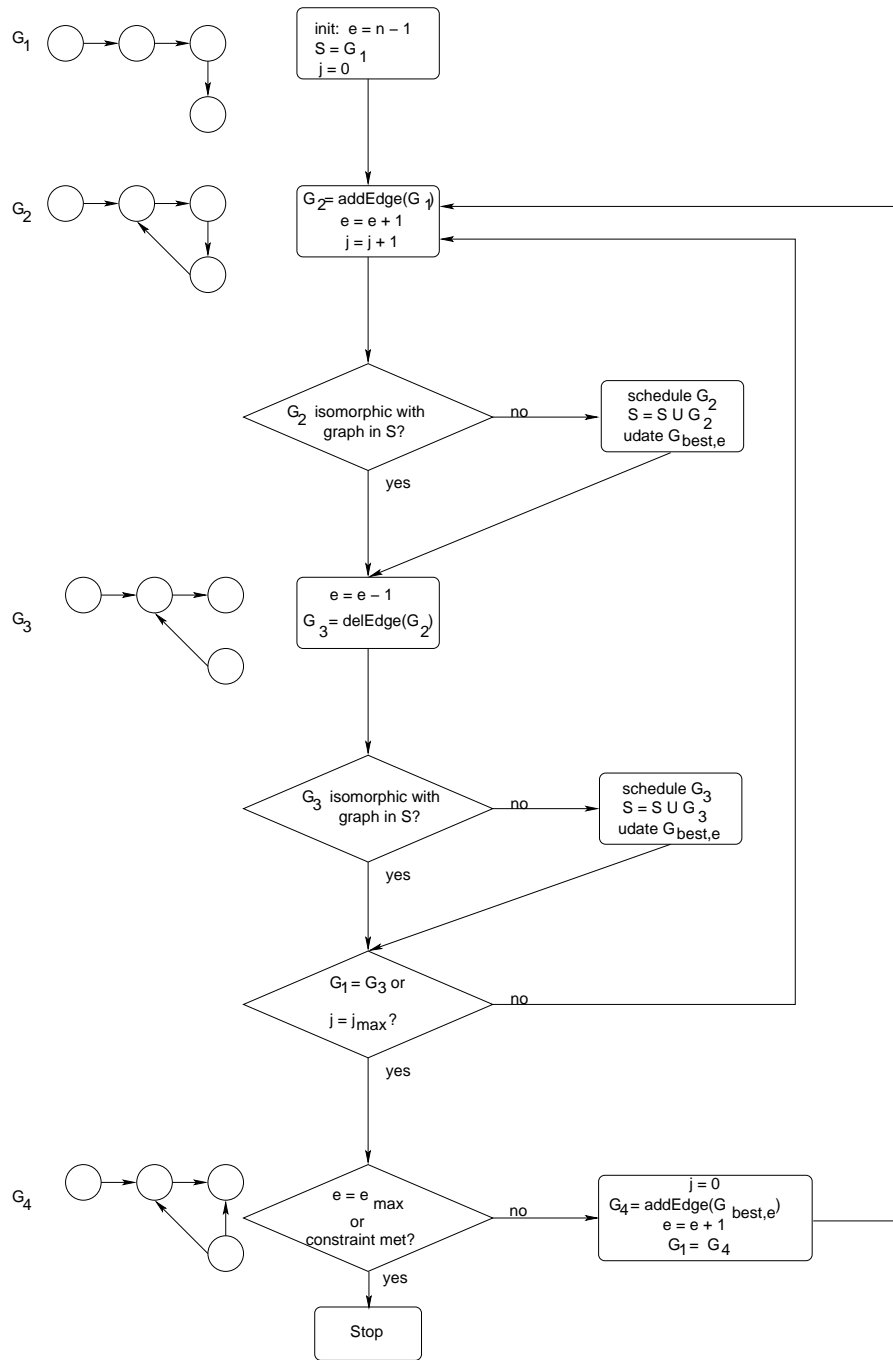


Figure 7.10: Bottom-up heuristic for constructing an efficient link topology that utilizes a graph isomorphism test.

the throughput using  $G_{\text{best},e}$  then we replace  $G_{\text{best},e}$ . Next we construct a graph  $G_3$  by removing an edge from  $G_2$ . If  $G_3$  is not isomorphic with a graph in  $S$  we schedule  $G_3$ , set  $S = S \cup \{G_3\}$ , and update  $G_{\text{best},e}$ . If  $G_3 = G_1$  then the algorithm is “stuck”—further combinations of adding and removing edges will produce graphs already evaluated. At this point we construct  $G_1$  by adding an edge to  $G_{\text{best},e}$  and repeat the above process by constructing a new  $G_2$  (with one more edge this iteration) by adding an edge to  $G_1$ . We continue until either the throughput constraint is met and the algorithm is successful, or until  $e = e_{\text{max}}$  and the algorithm fails to meet the throughput constraints with  $e_{\text{max}}$  edges.

The graph isomorphism test speeds up the deterministic link synthesis algorithm only if isomorphism testing of a topology graph is faster than the scheduling the application on the graph. The complexity of the graph isomorphism algorithm is still an open problem—there exists no known P algorithm for graph isomorphism testing, although the problem has also not been shown to be NP-complete. It is thought that the problem falls in the area between P and NP-complete, if such an area exists [99]. However, McKay’s *nauty* [78] program has been proven to be very efficient in practice. Although its worst case run time is exponential [81], an empirical test of a large number of randomly generated graphs produced run times of  $1.2p^2$  ns on a 1 GHz Pentium III machine where  $p$  is the number of nodes in the graph [78] ( $p$  equals the number of processors in our case). By comparison, the DLS scheduling algorithm has complexity  $O(v^3p)$  where  $v$  is the number of nodes in the *task graph* [97]. We modify the DLS scheduling algorithm by adding a flexibility calculation at each scheduling step. The complexity of the flexibility algorithm (Equation 6.1) is  $O(v(v+e)p \log p)$  where  $e$  is the number of edges in the graph, so the overall complexity scheduling an arbitrary graph using the modified

DLS scheduling algorithm is

$$O(v^4(v + e)p^2 \log p). \quad (7.9)$$

The number of tasks in the application will be much greater than the number of processors in practice, so  $v \gg p$  and  $e \gg p$ . For randomly generated graphs, the nauty program is therefore much faster than the modified DLS scheduling algorithm and we achieve significant speedup by detecting and exploiting graph isomorphism.

## Chapter 8

# Design Space Exploration Using Simulated Heating

### 8.1 Introduction

Application-specific, parameterized local search algorithms (PLSAs), in which optimization accuracy can be traded off with run-time, arise naturally in many optimization contexts, including most of the optimization problems discussed in this thesis. For many problems in system design, the user wishes to first quickly evaluate many trade-offs in the system, often in an interactive environment, and then to refine a few of the best design points as thoroughly as possible. Often, an exact system simulation may take days or weeks. In this context, it is quite useful to have optimization techniques where the run-time can be controlled, and which will generate a solution of maximum quality in the allotted time.

In this chapter we introduce a novel approach, which we call *simulated heating*, for systematically integrating parameterized local search into evolutionary algorithms (EAs). Using the framework of simulated heating, we investigate both static and dynamic strategies for systematically managing the trade-off between PLSA accuracy and

optimization effort. Our goal is to achieve maximum solution quality within a fixed optimization time budget.

We show that the simulated heating technique better utilizes the given optimization time resources than standard hybrid methods that employ fixed parameters, and that the technique is less sensitive to these parameter settings. In Chapter 9 we apply this framework to the voltage scaling optimization problem discussed in Section 4.3, a memory cost minimization problem for embedded systems, and the well-known binary knapsack problem. We compare our results to the standard hybrid methods, and show quantitatively that careful management of this trade-off is necessary to achieve the full potential of an EA/PLSA combination. We also explain how simulated heating could be used for the interconnect synthesis problem and for the problem of finding optimal transaction orders. Demonstrating the use of simulated heating on these last two problems is the subject of future work.

For many optimization problems, efficient algorithms exist for refining arbitrary points in the search space into better solutions. Such algorithms are called *local search algorithms* because they define neighborhoods, typically based on initial “coarse” solutions, in which to search for optima. Many of these algorithms are parameterizable in nature. Based on the values of one or more algorithm parameters, such a *parameterized local search algorithm (PLSA)* can trade off time or space complexity for optimization accuracy.

PLSAs and evolutionary algorithms (EAs) have complementary advantages. EAs are applicable to a wide range of problems, they are robust, and are designed to sample a large search space without getting stuck at local optima. Problem-specific PLSAs are often able to converge rapidly toward local minima. The term ‘local search’ generally applies to methods that cannot escape these minima. For these reasons, PLSAs can be

incorporated into EAs in order to increase the efficiency of the optimization.

Several techniques for incorporating local search have been reported. These include Genetic Local Search [80], Genetic Hybrids [40], Random Multi-Start [61], GRASP [38], and others. These techniques are often demonstrated on well-known problem instances where either optimal or near-optimal solutions are known. The optimization goal of these techniques is then to obtain a solution very close to the optimum with acceptable run-time. In this regard, the incorporation of local search has been quite successful. For example, Vasquez and Whitley [106] demonstrated results within 0.75% of the best known results for the Quadratic Assignment Problem using a hybrid approach, with all run times under five hours. In most of these hybrid techniques the local search is run with fixed parameter values (i.e. at the highest accuracy setting).

In this thesis, we consider a different optimization goal, which has not been addressed so far. Here we are interested in generating a solution of maximum quality within a specified optimization time, where the optimization run time is an important constraint that must be obeyed. Such a fixed optimization time budget is a realistic assumption in practical optimization scenarios. Many such scenarios arise in the design of embedded systems. In a typical design process, the designer begins with only a rough idea of the system architecture, and first needs to assess the effects of a large number of design choices—different component parts, memory sizes, different software implementations, etc. Since the time to market is very critical in the embedded system business, the design process is on a strict schedule. In the first phases of the design process, it is essential to get good estimates quickly so that these initial choices can be made. Later, as the design process converges on a specific hardware/software solution, it is important to get more accurate solutions. In these cases, the designer needs to have the run time as an input to the optimization problem.

In order to accomplish this goal, we vary the parameters of the local search during the optimization process in order to trade off accuracy for reduced complexity. Our optimization approach is general enough to hold for any kind of global search algorithm; however, in this paper we test hybrid solutions that solely use an EA as the global search algorithm. Existing hybrid techniques fix the local search at a single point, typically at the highest accuracy. In the following discussion and experiments, we refer to this method as a *fixed parameter method*. We will compare our results against this method.

One of the central issues we examine is how the computation time for the PLSA should be allocated during the course of the optimization. More time allotted to each PLSA invocation implies more thorough local optimization at the expense of a smaller number of achievable function evaluations (e.g., smaller numbers of generations explored with evolutionary methods), and vice-versa. Arbitrary management of this trade-off between accuracy and run time of the PLSA is not likely to generate optimal results. Furthermore, the proportion of time that should be allocated to each call of the local search procedure is likely to be highly problem-specific and even instance-specific. Thus, dynamic adaptive approaches may be more desirable than static approaches.

In this thesis, we describe a technique called *simulated heating* [12, 113], which systematically incorporates parameterized local search into the framework of global search. The idea is to increase the time allotted to each PLSA invocation during the optimization process—low accuracy of the PLSA at the beginning and high accuracy at the end<sup>1</sup>. This is in contrast to most existing hybrid techniques, which consider a fixed local search function, usually operating at the highest accuracy. Within the context of simulated heating optimization, we consider both static and dynamic strategies for

---

<sup>1</sup>In contrast to [113], the time budget here refers to the overall GSA/PLSA hybrid, not only the time resources needed by the PLSA.



systematically increasing the PLSA accuracy and the corresponding optimization effort. Our goals are to show that careful management of this trade-off is necessary to achieve the full potential of an EA/PLSA combination, and to develop an efficient strategy for achieving this trade-off management. We show that, in the context of a fixed optimization time budget, the simulated heating technique performs better than using a fixed local search.

In most heuristic optimization techniques, there are some parameters that must be set by the user. In many cases, there are no clear guidelines on how to set these parameters. Moreover, the optimal parameters are often dependent on the exact problem specification. We show that the simulated heating technique, while still requiring parameters to be set by the user, is less sensitive to the parameter settings.

First we will outline PLSAs for three of the optimization problems covered in this thesis.

### **8.1.1 PLSA for Voltage Scaling**

#### **Background**

Dynamic voltage scaling [73] in microprocessors is an important advancing technology. It allows the average power consumption in a device to be reduced by slowing down (by lowering the voltage) some tasks in the application. Here we will assume that the application is specified as a dataflow graph. We are given a schedule (ordering of tasks on the processors) and a constraint on the throughput of the system. We wish to find a set of voltages for all the tasks that will minimize the average power of the system while satisfying the throughput constraint. The only way to compute the throughput exactly in these systems is via a full system simulation. However, simulation is computationally intensive and we would like to minimize the number of simulations required during

synthesis. We have previously demonstrated that a data structure, called the *period graph*, can be used as an efficient estimator for the system throughput [9] and thus reduce the number of simulations required.

### Using the Period Graph for Local Search

As explained in [9] and in Chapter 4, we can estimate the throughput of the system as voltage levels are changed by calculating the maximum cycle mean<sup>2</sup> (MCM) [66] of the period graph. In order to construct the period graph, we must perform one full system simulation at an initial point—after the period graph is constructed we may use the MCM estimate without re-simulating the system. It is shown in [9] that the MCM of the period graph is an accurate estimate for the throughput if the task execution times are varied around a limited region (local search), and that the quality of the estimate increases as the size of this region decreases. A variety of efficient, low polynomial-time algorithms have been developed for computing the maximum cycle mean (e.g., see [29]).

We can use the size of the local search neighborhood as the parameter  $p$  in a parameterized local search algorithm (PLSA). We call this parameter the re-simulation threshold ( $r$ ), and define it as the vector distance between a candidate point (vector of voltages) and the voltage vector  $V$  from which the period graph was constructed. To search around a given point  $V$  in the design space, we must simulate once and build the period graph. Then, as long as the local search points are within a distance  $r$  from  $V$ , we can use the (efficient) period graph estimate. For points outside  $r$ , we must re-simulate

---

<sup>2</sup>Here the maximum cycle mean is the maximum, over all directed cycles of the period graph, of the sum of the task execution times on a cycle divided by the sum of the edge delays (initial tokens) on a cycle.

and rebuild the period graph. Consequently, there is a trade-off between speed and accuracy for  $r$ —as  $r$  decreases, the period graph estimate is more accurate, but the local search is slower since simulation is performed more often.

### **PLSA Implementation**

To solve the dynamic voltage scaling optimization problem we use a GSA/PLSA hybrid where an evolutionary algorithm is the GSA and the PLSA is either a hill climbing [64] or Monte Carlo [57] search utilizing the period graph. Pseudo-code for both local search methods is shown in Figures 8.1 and 8.2. The benefit of using a local search algorithm is that within a restricted voltage range we can use the period graph estimator for the throughput, which is much faster than performing a simulation. The local search algorithms are explained further below.

#### **Voltage Scaling PLSA 1: Hill Climb Local Search**

For the hill climbing algorithm, we defined a parameter  $\delta$ , which is the voltage step, and a re-simulation threshold  $r$ , which is the maximum amount that the voltage vector can vary from the point at which the period graph was calculated. We ran the algorithm for  $I$  iterations. So for this case, the PLSA  $L$  had 3 parameters  $I$ ,  $r$ , and  $\delta$ . One iteration of local search consisted of changing the node voltages, one at a time, by  $\pm\delta$ , and choosing the direction in which the objective function was minimized. From this, the worst case cost  $C(I, r, \delta)$  for  $I$  iterations would correspond to evaluating the objective function  $3I$  times, and re-simulating  $(I/\lceil r/\delta \rceil)$  times. For our experiments we fixed  $I$  and  $\delta$  and defined the local search parameter as  $p = 1/r$ . Then for smaller  $p$  (corresponding to larger re-simulation threshold) the voltage vector can move a greater distance before a new simulation is required. For a fixed number of iterations  $I$  in the local search, a

**Algorithm 8.1:** HILL CLIMB LOCAL SEARCH(

$\vec{V}_{\text{in}}, N, I, G, F_{\text{obj}}, \delta_{\text{resim}}, \vec{V}_{\text{out}}, \text{score}$

)

**input:** voltage vector  $\vec{V}_{\text{in}}$  of size  $N$

**input:** number of iterations  $I$

**input:** period graph  $G$  with  $N$  tasks

**input:** objective function  $F_{\text{obj}}$  derived from maximum cycle mean of  $G$  scaled by  $\vec{V}$

**input:**  $\delta_{\text{resim}}$  is the resimulation threshold distance

**output:** voltage vector  $\vec{V}_{\text{out}}$

LowScore  $\leftarrow \infty$

$\delta \leftarrow \delta_{\text{resim}}/100$

$\vec{V} \leftarrow \vec{V}_{\text{in}}$

**for** ( $k = 0; k < I; k++$ )

**do** {

**for** ( $i = 0; i < N; i++$ )

{

$V_0 \leftarrow V[i]$

$V[i] \leftarrow V_0(1 + \delta)$

$f_1 \leftarrow F_{\text{obj}}(\vec{V}, G)$

$V[i] \leftarrow V_0(1 - \delta)$

$f_2 \leftarrow F_{\text{obj}}(\vec{V}, G)$

$V[i] = V_0$

$f \leftarrow F_{\text{obj}}(\vec{V}, G)$

**do** {

**if** ( $f_1 < f$ )

**then** {  $V[i] \leftarrow V_0(1 + \delta)$

**else if** ( $f_2 < f$ )

**then** {  $V[i] \leftarrow V_0(1 - \delta)$

$D \leftarrow \|\vec{V} - \vec{V}_{\text{in}}\|$

**if** ( $D < \delta_{\text{resim}}$ )

**then** { Resimulate and rebuild  $G$

$\vec{V}_{\text{in}} \leftarrow \vec{V}$

score  $\leftarrow F_{\text{obj}}(\vec{V}, G)$

**if** (score < LowScore)

**then** { LowScore  $\leftarrow$  score

$\vec{V}_{\text{out}} \leftarrow \vec{V}$

}

}

}

Figure 8.1: Pseudo-code for hill climb local search for voltage scaling application.

**Algorithm 8.2:** MONTE CARLO LOCAL SEARCH(

$\vec{V}_{\text{in}}, N, R, G, F_{\text{obj}}, D, \delta_{\text{resim}}, \vec{V}_{\text{out}}, \text{score}$

)

**input:** voltage vector  $\vec{V}_{\text{in}}$  of size  $N$

**input:** number of random points generated  $R$

**input:**  $D$  is maximum distance from  $\vec{V}_{\text{in}}$  to random point

**input:** period graph  $G$  with  $N$  tasks

**input:** objective function  $F_{\text{obj}}$  derived from maximum cycle mean of  $G$  scaled by  $\vec{V}$

**input:**  $\delta_{\text{resim}}$  is the resimulation threshold distance

**output:** voltage vector  $\vec{V}_{\text{out}}$

**output:** score

**Generate** a list  $L_{\text{rand}}$  of  $R$  random vectors uniformly distributed within a distance no more than  $D$  from  $\vec{V}_{\text{in}}$

$\vec{V}_x \leftarrow \vec{V}_{\text{in}}$

score  $\leftarrow \infty$

**for** ( $i = 1$  to  $R$ )

**do**  $\{q_r \leftarrow \|\vec{V}_r - \vec{V}_{\text{in}}\|$

**while** ( $L_{\text{rand}}$  not empty )

**do**  $\left\{ \begin{array}{l} \text{Pop head of list } L_{\text{rand}} \text{ to get } \vec{V} \\ \text{Scale } G \text{ by } \vec{V} \\ q \leftarrow \|\vec{V}_x - \vec{V}\| \\ \text{if } (q < \delta_{\text{resim}}) \end{array} \right.$

**then**  $\left\{ \begin{array}{l} f \leftarrow F_{\text{obj}}(\vec{V}, G) \\ \text{if } (f < \text{score}) \\ \quad \text{then } \{ \text{score} \leftarrow f \end{array} \right.$

**else**  $\left\{ \begin{array}{l} \vec{V}_x \leftarrow \vec{V} \\ \text{Resimulate around } \vec{V} \text{ and rebuild } G \\ \text{for } (r = 1 \text{ to } \text{size}(L_{\text{rand}})) \\ \quad \text{do } \{q_r \leftarrow \|\vec{V}_r - \vec{V}_x\| \\ \text{Sort } L_{\text{rand}} \text{ according to lowest } q \text{ first} \end{array} \right.$

Figure 8.2: Pseudo-code for Monte Carlo local search for voltage scaling application.

smaller  $p$  corresponds to a shorter running time  $C(p)$  for  $L(p)$ . The accuracy  $A(p)$  is lower, since the accuracy of the period graph estimate decreases as the voltage vector moves farther away from the simulation point.

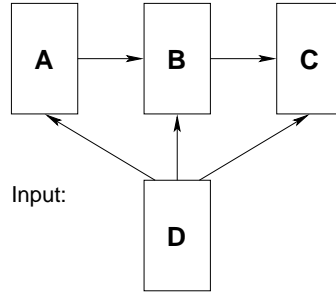
### **Voltage Scaling PLSA 2: Monte Carlo Local Search**

In the Monte Carlo algorithm, we generated  $N$  random voltage vectors within a distance  $D$  from the input vector. For all points within a re-simulation threshold  $r$ , we used the period graph to estimate performance. A greedy strategy was used to evaluate the remaining points. Specifically, we selected one of the remaining points at random, performed a simulation to construct a new period graph, and used the resulting estimator to evaluate all points within a distance  $r$  from this point. If there were points remaining after this, we chose one of these and repeated the process. For the experiments we fixed  $N$  and  $D$  and defined local search parameter  $p = 1/r$ . As for the hill climbing local search, smaller values of  $p$  correspond to shorter run times and less accuracy for the Monte Carlo local search.

### **8.1.2 PLSA for Interconnect Synthesis**

In Section 7.1 we described a greedy heuristic algorithm, called the TPLA algorithm [6], to synthesize an interconnect and an associated multiprocessor schedule for a given application. Here we will describe how this algorithm can be parameterized so that it can be used as a PLSA for simulated heating.

The TPLA algorithm starts with a fully connected network, and operates in *down* and *up* phases. Each step of the down phase in TPLA removes one link, while each step of the up phase adds one link. We can modify this basic idea to create a parameterized local search for interconnect synthesis. The input to the local search is a processor



Remove (A,B) and (B,C)

Add (B,A) and (C,D)

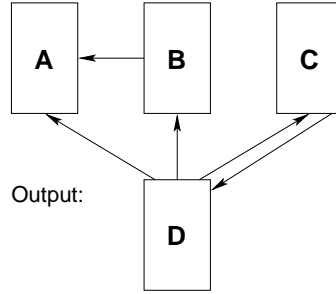


Figure 8.3: PLSA for interconnect synthesis. The output topology graph is one possible topology generated from the input topology graph with  $\rho = 2$ .

topology graph (Section 6.1.1)  $T(\Phi, L)$  with  $\phi$  processors (nodes in the topology graph) and  $l$  links (edges in the topology graph). We first remove  $\rho$  links, where  $\rho \leq l$ . There are  $d = \binom{l}{\rho}$  possible choices. Next we add back  $\rho$  links to achieve a new topology with  $l$  links. This effectively creates a set of topologies in a local region around the input topology. Figure 8.3 illustrates this concept.

There are  $\phi(\phi - 1) - (l - \rho)$  positions where a link may be added where one does not already exist, so there are  $u = \binom{\phi(\phi-1)-(l-\rho)}{\rho}$  possible choices for adding back the  $\rho$  links. The total number of combinations for first removing  $\rho$  links then adding back  $\rho$  links is the product of  $u$  and  $d$ . Many of these topologies may be isomorphic to one

another, so the online graph isomorphism test (Section 7.3) can be used to avoid evaluating isomorphic topologies. Each time an isomorphically unique topology is created, it is evaluated using a scheduling algorithm utilizing the feasibility and flexibility metrics (Chapter 6), and the topology with the best schedule is chosen as the output of the local search.

The number of topologies generated (the local search complexity) is a rapidly increasing function of  $\rho$ , and  $\rho$  can be used as the PLSA parameter  $p$ . It is also possible to place a limit on the number of combinations of  $\rho$  links removed ( $d_{\max}$ ) and the number of combinations of  $\rho$  links added back ( $u_{\max}$ ). In this case the local search parameter  $p = d_{\max}u_{\max}$  can be adjusted so that the local search complexity does not increase so fast with increasing  $p$ .

### 8.1.3 PLSA for Ordered Transactions

The ordered transactions strategy was covered in Chapter 5, where it was shown that the problem of finding optimal transaction orders is NP-complete. In this section, we outline how a PLSA could be constructed for this problem. Recall that the ordered transactions graph (Section 5.1.3)  $\Gamma(G_{\text{IPC}}, O)$  is created from an IPC graph  $G_{\text{IPC}}$  and a transaction order  $O$ , and that the MCM of  $\Gamma$  gives the throughput of the system. A PLSA for the ordered transactions problem takes an input ordering  $O_{\text{in}}$  and evaluates permutations around  $O_{\text{in}}$  to produce a better ordering  $O_{\text{out}}$ . The permutation method we propose is a pair swap—we swap the positions of a pair of nodes in the transaction ordering. If the swapping does not create any zero-delay cycles, we can calculate the MCM of the new ordered transaction graph. If the pair swap has produced a lower MCM, we keep this ordering and attempt to swap another pair of nodes. This continues for a number  $p$  of iterations, where  $p$  is the local search parameter. Pseudo-code for this local search is



**Algorithm 8.3:** ORDERED TRANSACTIONS LOCAL SEARCH(

$O_{in}, G_{IPC}, O_{out}$

)

**input:** transaction ordering  $O_{in}$  of length  $n$

**input:** IPC graph  $G_{IPC}$

**output:** new transaction ordering  $O_{out}$

$i \leftarrow n$

$O_{out} \leftarrow O_{in}$

$bestScore \leftarrow MCM(\Gamma(G_{IPC}, O_{in}))$

$count \leftarrow 0$

**while**  $(i > 0) \wedge (count < pn)$

$\left. \begin{array}{l} \text{do} \end{array} \right\}$	$\left. \begin{array}{l} \text{do} \end{array} \right\}$	$\left. \begin{array}{l} \text{then} \end{array} \right\}$	$j = -1$
			$i = i - 1$
			<b>while</b> $(j < i) \wedge (count < pn)$
			$\left. \begin{array}{l} j \leftarrow j + 1 \\ \text{if } O_{out}[i] \neq O_{out}[j] \\ \quad \left. \begin{array}{l} \text{temp} \leftarrow O_{out}[i] \\ O_{out}[i] \leftarrow O_{out}[j] \\ O_{out}[j] \leftarrow \text{temp} \end{array} \right\} \\ \quad \text{if zero-delay cycles in } \Gamma(G_{IPC}, O_{out}) \\ \quad \quad \left. \begin{array}{l} \text{then } \{score \leftarrow \infty\} \\ \text{else } \{score \leftarrow MCM(\Gamma(G_{IPC}, O_{out}))\} \end{array} \right\} \\ \quad \text{if } score < bestScore \\ \quad \quad \left. \begin{array}{l} \text{then } \{bestScore \leftarrow score\} \\ \text{else } \left\{ \begin{array}{l} \text{temp} \leftarrow O_{out}[i] \\ O_{out}[i] \leftarrow O_{out}[j] \\ O_{out}[j] \leftarrow \text{temp} \end{array} \right\} \end{array} \right\} \end{array} \right\}$

Figure 8.4: Pseudo-code for PLSA for ordered transactions strategy.

given in Figure 8.4

## 8.2 Hybrid Global/Local Search Related Work

In the field of evolutionary computation, hybridization seems to be common for real-world applications [43] and many evolutionary algorithm/local search method combinations can be found in the literature, e.g., [30, 53, 80, 92, 111]. Local search techniques can often be incorporated naturally into evolutionary algorithms (*EAs*) in order to increase the effectiveness of optimization. This has the potential to exploit the com-

plementary advantages of EAs (generality, robustness, global search efficiency), and problem-specific PLSAs (exploiting application-specific problem structure, rapid convergence toward local minima). Below we list some hybrid methods in the literature, and suggest how they could potentially be adapted to use our simulated heating technique.

One problem to which hybrid approaches have been successfully applied is the quadratic assignment problem (QAP), which is an important combinatorial problem. Several groups have used hybrid genetic algorithms that are effective in solving the QAP. The QAP concerns  $n$  facilities, which must be assigned to  $n$  locations at minimum cost. The problem is to minimize the cost

$$C(\pi) = \sum_{i=1}^n \sum_{j=1}^n a_{ij} b_{\pi(i)\pi(j)}, \quad \pi \in \Pi(n)$$

where  $\Pi(n)$  is a set of all permutations of  $\{1, 2, \dots, n\}$ ,  $a_{ij}$  are elements of a distance matrix, and  $b_{ij}$  are elements of a flow matrix representing the flow of materials from facility  $i$  to facility  $j$ .

Merz and Freisleben [80] presented a Genetic Local Search (GLS) technique, which applies a variant of the *2-opt* heuristic as a local search technique. For the QAP, the *2-opt* neighborhood is defined as the set of all solutions that can be reached from the current solution by swapping two elements of the permutation  $\pi$ . The size of this neighborhood increases quadratically with  $n$ . The *2-opt* local search employed by Merz takes the first swap that reduces the total cost  $C(\pi)$ . This is done to increase efficiency.

Fleurent and Ferland [40] combined a genetic algorithm with a local Tabu Search (TS) method. In contrast to the simpler local search of Merz, the idea of the TS is to consider all possible moves from the current solution to a neighboring solution. Their method is called Genetic Hybrids. They improved the best solutions known at the time for most large scale QAP problems.

By comparison, simulated heating for QAP might be formulated as a combination

of the above two methods. One could consider the best of  $m$  moves found that reduce  $C(\pi)$ , where  $m$  is the PLSA parameter.

Vasquez and Whitley [106] also presented a technique, which combines a genetic algorithm with TS, where the genetic algorithm is used to explore in parallel several regions of the search space and uses a fixed Tabu local search to improve the search around some selected regions. They demonstrated near optimal performance, within 0.75% of the best known solutions. They did not investigate their technique in the context of a fixed optimization time budget.

Random multi-start local search has been one of the most commonly used techniques for combinatorial optimization problems [61, 91]. In this technique, a number of solutions are generated randomly at each step, local search is repeated on these solutions, and the best solution found during the entire optimization is output. Several improvements over random multi-start have been described. Greedy randomized adaptive search procedures (GRASP) combine the power of greedy heuristics, randomization, and conventional local search procedures [38]. Each GRASP iteration consists of two phases—a construction phase and a local search phase. During the construction phase, each element is selected at random from a list of candidates determined by an adaptive greedy algorithm. The size of this list is restricted by parameters  $\alpha$  and  $\beta$ , where  $\alpha$  is a value restriction and  $\beta$  is a cardinality restriction. Feo et al. demonstrate the GRASP technique on a single machine scheduling problem [39], a set covering problem, and a maximum independent set problem [38]. They run the GRASP for several fixed values of  $\alpha$  and  $\beta$ , and show that the optimal parameter values are problem dependent. In simulated heating,  $\alpha$  and  $\beta$  would be candidates for parameter adaptation. In the second phase of GRASP, a local search is applied to the constructed solution to find a local optimum. For the set covering problem, Feo et al. define a  $k, p$  exchange local search

where all  $k$ -tuples in a cover are exchanged with a  $p$ -tuple. Here,  $k$  was fixed during optimization. In a simulated heating optimization,  $k$  might be used as the PLSA parameter, with smaller tuples being exchanged at the beginning of the optimization and larger tuples examined at the end. A similar  $k$ -exchange local search procedure was used for the maximum independent set problem.

Kazarlis et al. [60] demonstrate a microgenetic algorithm (MGA) as a generalized hill-climbing operator. The MGA is a GA with a small population and a short evolution. The main GA performs global search while the MGA explores a neighborhood of the current solution provided by the main GA, looking for better solutions. The main advantage of the MGA is its ability to identify and follow narrow ridges of arbitrary direction leading to the global optimum. Applied to simulated heating, MGA could be used as the local search function with the population size and number of generations used as PLSA parameters.

He and Xu [49] describe three hybrid genetic algorithms for solving linear and partial differential equations. The hybrid algorithms integrate the classical successive over relaxation (SOR) with evolutionary computation techniques. The recombination operator in the hybrid algorithm mixes two parents, while the mutation operator is equivalent to one iteration of the SOR method. A relaxation parameter  $\omega$  for the SOR is adapted during the optimization. He and Xu observe that is very difficult to estimate the optimal  $\omega$ , and that the SOR is very sensitive to this parameter. Their hybrid algorithm does not require the user to estimate the parameter; rather, it is evolved during the optimization. Different relaxation factors are used for different individuals in a given population. The relaxation factors are adapted based on the fitness of the individuals. By contrast, in simulated heating all members of a given population are assigned the same local search parameter at a given point in the optimization.

When employing PLSAs in the context of many optimization scenarios, however, a critical issue is how to use computational resources most efficiently under a given optimization time budget (e.g., a minute, an hour, a day, etc.). Goldberg and Voessner [44] study this issue in the context of a fixed local search time. They idealize the hybrid as consisting of steps performed by a global solver  $G$ , followed by steps by a local solver  $L$ , and a search space as consisting of basins of attraction that lead to acceptable targets. Using this, they are able to decompose the problem of hybrid search, and to characterize the optimum local search time that maximizes the probability of achieving a solution of a specified accuracy.

Here, we consider both fixed and variable local search time. The issue of how to best manage computational resources under a fixed time budget translates into a problem of appropriately reconfiguring successive PLSA invocations to achieve appropriate accuracy/run-time trade-offs as optimization progresses.

### 8.3 Simulated Heating

From the discussion of prior work we see that one weakness of many existing approaches is their sensitivity to parameter settings. Also, excellent results have been achieved through hybrid global/local optimization techniques, but they have not been examined carefully for a fixed optimization time budget. In the context of a limited time budget, we are especially interested in minimizing wasted time. One obvious place to focus is at the beginning of the optimization, where many of the candidate solutions generated by the global search are of poor quality. Intuitively, one would want to evaluate these initial solutions quickly and not spend too much time on the local search. Also, it is desirable to reduce the number of trial runs required to find an optimal parameter

setting. One way to do this is to require only that a good *range* for the parameter be given. These considerations lead to the idea of simulated heating.

### 8.3.1 Basic Principles

A general single objective optimization problem can be described as an objective function  $f$  that maps a tuple of  $m$  parameters (decision variables) to a single objective  $y$ . Formally, we wish to either minimize or maximize  $y = f(\vec{x})$  subject to  $\vec{x} = (x_1, x_2, \dots, x_m) \in X$  where  $\vec{x}$  is called the *decision vector*,  $X$  is the *parameter space* or *search space*, and  $y$  is the objective. A solution candidate consists of a particular  $(y_0, \vec{x}_0)$  where  $y_0 = f(\vec{x}_0)$ .

We will approach the optimization problem by using an *iterative search process*. Given a set  $X$ , and a function  $F$ , which maps  $X$  onto itself, we define an iterative search process as a sequence of successive approximations to  $F$ , starting with an  $x^0$  from  $X$ , with  $x^{r+1} = F(x^r)$  for  $r = (0, 1, 2, \dots)$ . One *iteration* is defined as a consecutive determination of one candidate from another candidate set using some  $F$ . For an evolutionary algorithm, one iteration consists of the determination of one generation from the previous generation, with  $F$  consisting of the selection, crossover, and mutation rules.

The basic idea behind simulated heating is to vary the local search parameter  $p$  during the optimization process. This is in contrast to the more commonly employed technique of choosing a single value for  $p$  (typically that value producing highest accuracy of the local search  $L(p)$ ) and keeping it constant during the entire optimization. Here, we start with a low value for  $p$ , which implies a low cost  $C(p)$ , and accuracy  $A(p)$  for the local search, and increase  $p$  at certain points in time during the optimization, which increases  $C(p)$  and  $A(p)$ . This is depicted in Figure 8.5, where the dotted line corresponds to simulated heating, and the dashed line corresponds to the traditional ap-

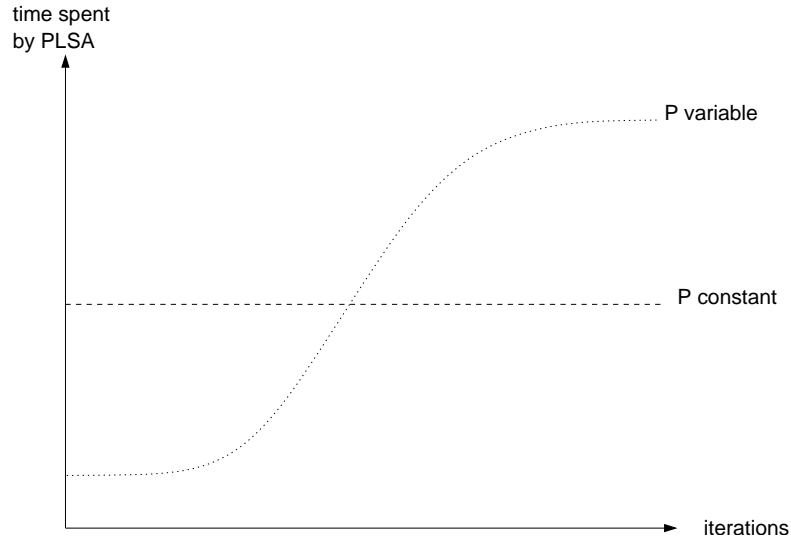


Figure 8.5: Simulated heating vs. traditional approach to utilizing local search.

proach. The goal is to focus on the global search at the beginning and to find promising regions of the search space first; for this phase,  $L(p)$  runs with low accuracy, which in turn allows a greater number of optimization steps of the global search  $G$ . Afterwards, more time is spent by  $L(p)$  in order to improve the solutions found or to assess them more accurately. As a consequence, fewer global search operations are possible during this phase of optimization. Since  $A(p)$  is systematically increased during the process, we use the term *simulated heating* for this approach by analogy to simulated annealing where the ‘temperature’ is continuously decreased according to a given cooling scheme.

### 8.3.2 Optimization Scenario

We assume that we have a global search algorithm (GSA)<sup>3</sup>  $G$  operating on a set of solution candidates and a PLSA  $L(p)$ , where  $p$  is the parameter of the local search

---

<sup>3</sup>In this thesis, we focus on an evolutionary algorithm as the global search algorithm, although the approach is general enough to hold for any global search algorithm.

procedure<sup>4</sup>. Let

- $C_{\text{fix}}$  define the maximum (worst-case) time needed by  $G$  to generate a new solution that is inserted in the next solution candidate set,
- $C(p)$  denote the complexity (worst-case run-time) of  $L$  for the parameter choice  $p$ ,
- $A(p)$  be the accuracy (effectiveness) of  $L$  with regard to  $p$ , and
- $R$  denote the set of permissible values for parameter  $p$ . Typically,  $R$  may be described by an interval  $[p_{\min} \dots p_{\max}] \cap \Re$  where  $\Re$  denotes the set of reals and  $C(p_{\min}) \leq C(p_{\max})$ .

Furthermore, suppose that for any pair  $(p_1, p_2)$  of parameter values we have that

$$(p_1 \leq p_2) \implies (C(p_1) \leq C(p_2)) \textbf{ and } (A(p_1) \leq A(p_2)) \quad (8.1)$$

That is, increasing parameter values in general result in increased consumption of compile-time, as well as increased optimization effectiveness.

Generally, it is very difficult, if not impossible, to analytically determine the functions  $C(p)$  and  $A(p)$ , but these functions are useful conceptual tools in discussing the problem of designing cooperating GSA/PLSA combinations. The techniques that we explore in this thesis do not require these functions to be known. The only requirement we make is that the monotonicity property 8.1 be obeyed at least in an approximate sense (fluctuations about relatively small variations in parameter values are admissible, but significant increases in the PLSA parameter value should correspond to increasing cost and accuracy). Consequently, a tunable trade-off emerges: when  $A(p)$  is low, refinement is generally low as well, but not much time is consumed ( $C(p)$  is also low).

---

<sup>4</sup>For simplicity it is assumed here that  $p$  is a scalar rather than a vector of parameters.



Conversely, higher  $A(p)$  requires higher computational cost  $C(p)$ . We define simulated heating as follows:

**Definition 1: [Heating scheme]**

A heating scheme  $H$  is a triple  $H = (H_R, H_{it}, H_{set})$  where:

- $H_R$  is a vector of PLSA parameter values with  $H_R = (p_1, \dots, p_n)$ ,  
 $p_i \in [p_{\min}, \dots, p_{\max}]$ , and  $p_1 \leq p_2 \leq \dots \leq p_n$ ,
- $H_{it}$  is a boolean function, which yields true if the number of iterations performed for parameter  $p_i$  does not exceed the maximum number of iterations allowed for  $p_i$ , and
- $H_{set}$  is a boolean function, which yields true if the size of the solution candidate set does not exceed the maximum size for  $p_i$  and iteration  $t$  of the overall GSA/PLSA hybrid.

The meanings of the functions  $H_{it}$  and  $H_{set}$  will become clear in the global/local hybrid algorithm of Figure 8.6, which is taken as the basis for the optimization scenario considered in this thesis.

The GSA considered here is an evolutionary algorithm (EA) that is

1. Generational, i.e., at each evolution step an entirely new population is created. This is in contrast to a non-generational or steady-state EA that only considers a single solution candidate per evolution step;
2. Baldwinian, i.e., the solutions improved by the PLSA are not re-inserted in the population. This is in contrast to a Lamarckian EA, in which solutions would be updated after PLSA refinement.

**Algorithm 8.1: Global/Local Hybrid()**

**Input:**  $H = ((p_1, \dots, p_n), H_{\text{it}}, H_{\text{set}})$  (heating scheme)  
 $T_{\text{max}}$  (maximum time budget)

**Output:**  $s$  (best solution found)

**Step 1: Initialization:** Set  $T = 0$  (time used),  $t = 0$  (iterations performed), and  $i = 1$  (current PLSA parameter index).

**Step 2: Heating:** Set  $p = p_i$ .

**Step 3: Next iteration:** Create an empty multi-set of solution candidates  $S_t = \emptyset$ .

**Step 4: Global search:** If  $t = 0$ , create a solution candidate  $s$  at random. Otherwise, generate a new solution candidate using  $G$  based on the previous solution candidate set  $S_{t-1}$  and the associated quality function  $F_{t-1}$ .

**Step 5: Local search:** Apply  $L$  with parameter  $p$  to  $s$  and assign it a quality (fitness)  $F_t(s)$ .

**Step 6: Termination for candidate set:** Set  $S_t = S_t + s$  and  $T = T + C_{\text{fix}} + C(p)$ . If the condition  $H_{\text{set}}$  is fulfilled and  $T \leq T_{\text{max}}$  then go to Step 4.

**Step 7: Termination for iteration:** Set  $t = t + 1$ . If the condition  $H_{\text{it}}$  is fulfilled and  $T \leq T_{\text{max}}$  then go to Step 3.

**Step 8: Termination for algorithm:** If  $i < n$  increment  $i$ . If  $T \leq T_{\text{max}}$  then go to Step 2.

**Step 9: Output:** Apply  $L$  with parameter  $p_{\text{max}}$  to the best solution in  $\bigcup_{1 \leq i \leq t} S_t$  regarding the corresponding quality functions  $F_i$ ; the resulting solution  $s$  is the outcome of the algorithm.

Figure 8.6: Global/Local Search Hybrid.

## 8.4 Simulated Heating Schemes

We are interested in exploring optimization techniques in which the overall optimization time is fixed and specified in advance (fixed time budget). During the optimization and within this time budget, we allow a heating scheme to adjust three optimization parameters per PLSA parameter value:

1. the number of GSA iterations  $t_p$ ,
2. the size of the solution candidate set  $N_i$ , and
3. the maximum optimization time using this parameter value  $T_i$ .

We distinguish between static and dynamic heating based on how many of the parameters are fixed and how many are allowed to vary during the optimization. This is illustrated in Figure 8.7. In our experiments, we keep the size of the solution candidate (GA population) fixed, and thus only consider the FIS, FTS, and VIT strategies. For the sake of completeness, however, we outline all these strategies below.

### 8.4.1 Static Heating

Static heating means that at least two of the above three parameters are fixed and identical for all PLSA parameter values considered during the optimization process. As a consequence, the third parameter is either given as well or can be calculated before run-time for each PLSA parameter value separately. As illustrated in Figure 8.7 on the left, there are four possible static heating schemes.

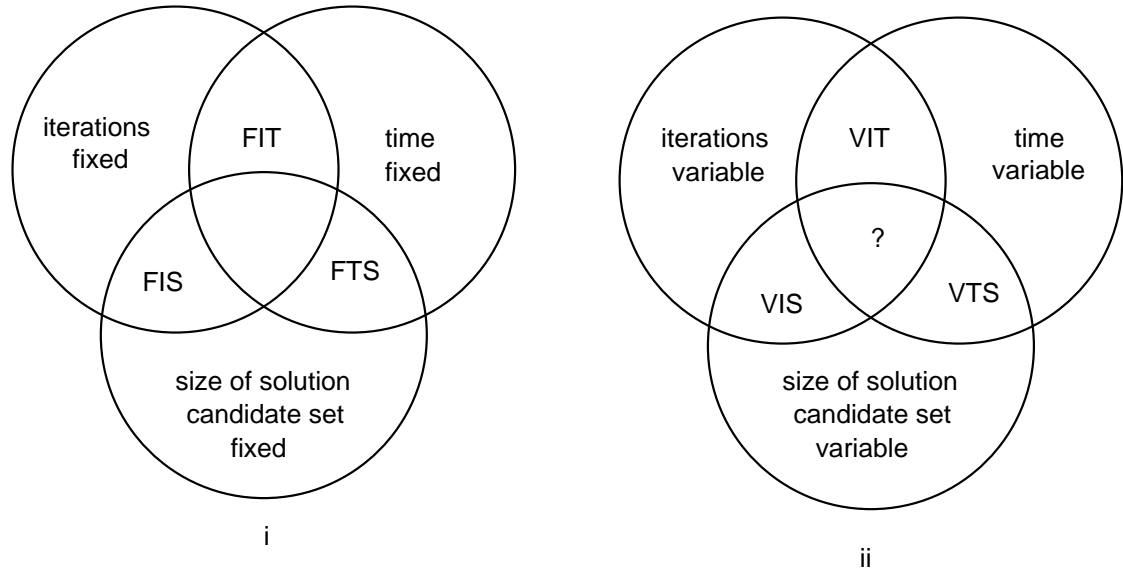


Figure 8.7: Illustration of the different types of *i*) static heating and *ii*) dynamic heating. For static heating, at least two of the three attributes are fixed. (**FIS** refers to **fixed iterations** and population size per parameter; **FTS** refers to **fixed time** and population **size** per parameter; **FIT** refers to **fixed iterations** and fixed **time** per parameter.) For dynamic heating, at least two attributes are variable. (**VIT** refers to **variable iterations** and **time** per parameter; **VIS** refers to **variable iterations** and population **size**; **VTS** refers to **variable time** and population **size**. In our experiments, we will only consider the FIS, FTS, and VIT strategies.

### PLSA Parameter Fixed — Standard Hybrid Approach

Fixing all three parameters is identical to keeping  $p$  constant. Thus, only a single PLSA parameter value is used during the optimization process. This scheme represents the common way to incorporate PLSAs into GSAs and is taken as the reference for the other schemes as actually no heating is performed.

### Number of Iterations and Size of Solution Candidate Set Fixed Per PLSA Parameter (FIS)

In this strategy (FIS), the parameter  $p_i$  is constant for exactly  $t_i = t_p$  iterations. The question is, therefore, how many iterations  $t_p$  may be performed per parameter within the time budget  $T_{\max}$ . Having the constraint

$$T_{\max} \geq t_p N(C_{\text{fix}} + C(p_1)) + t_p N(C_{\text{fix}} + C(p_2)) + \dots + t_p N(C_{\text{fix}} + C(p_n))$$

we obtain  $t_p$  with

$$t_p = \left\lfloor \frac{T_{\max}}{N \sum_{i=1}^n (C_{\text{fix}} + C(p_i))} \right\rfloor \quad (8.2)$$

as the number of iterations assigned to each  $p_i$ .

### Amount of Time and Size of Solution Candidate Set Fixed Per PLSA Parameter (FTS)

For the FTS strategy, the points in time where  $p$  is increased are equi-distant and may be simply computed as follows. Obviously the time budget, when equally split between  $n$  parameters, becomes  $T_p = T_{\max}/n$  per parameter. Hence, the number of iterations  $t_i$  that may be performed using parameter  $p_i$ ,  $i = 1, \dots, n$  is restricted by

$$t_i N(C_{\text{fix}} + C(p_i)) \leq T_p, \forall i = 1, \dots, n$$

Thus, we obtain

$$t_i = \left\lfloor \frac{T_{\max}}{nN(C_{\text{fix}} + C(p_i))} \right\rfloor \quad (8.3)$$

as the maximum number of iterations that may be computed using parameter  $p_i$  in order to stay within the given time budget.

### Number of Iterations and Amount of Time Fixed Per PLSA Parameter (FIT)

With the FIT scheme the size of the solution candidate set is different for each PLSA parameter considered. The time per iteration for parameter  $p_i$  is given by  $T_i = T_{\max}/t_{\max}$  and is the same for all  $p_i$  with  $1 \leq i \leq n$ . This relation together with the constraint

$$T_i \geq N_i(C_{\text{fix}} + C(p_i))$$

yields

$$N_i = \left\lfloor \frac{T_{\max}}{t_{\max}(C_{\text{fix}} + C(p_i))} \right\rfloor \quad (8.4)$$

as the maximum size of the solution candidate set for  $p_i$ .

### 8.4.2 Dynamic Heating

In contrast to static heating, dynamic heating refers to the case in which at least two of the three optimization parameters are not fixed and may vary for different PLSA parameters. The four potential types of dynamic heating are shown in Figure 8.7. However, the scenario where all three optimization parameters are variable and may be different for each PLSA parameter is more hypothetical than realistic. This approach is not investigated in this thesis and only listed for reasons of completeness. Hence, we consider three dynamic heating schemes where only one parameter is fixed. One of the variable parameters is determined dynamically during run-time according to a predefined criterion. Here, the criterion is whether an improvement with regard to the solutions

generated can be observed during a certain time interval (measured in seconds, number of solutions generated, or number of iterations performed). The time constraint is defined in terms of the remaining variable parameter.

### **Number of Iterations and Size of Solution Candidate Set Variable Per PLSA Parameter (VIS)**

With the VIS strategy, the time  $T_i = T_{\max}/n$  per PLSA parameter value is fixed (and identical for all  $p_i$ ). If the time constraint is defined on the basis of the number of solutions generated, the hybrid works as follows: As long as the time  $T_i$  is not exceeded, new solutions are generated using  $p_i$  and copied to the next solution candidate set—otherwise, the next GSA iteration with  $p_{i+1}$  is performed. If, however, the time elapsed for the current iteration is less than  $T_i$  and none of the recently generated  $N_{\text{stag}}$  solutions achieves an improvement in fitness, the next iteration with  $p_i$  is started.

It is not practical to consider a certain number of iterations as the time constraint—since the time per iteration is not known, there is no condition that determines when the filling of the next solution candidate set can be stopped.

### **Amount of Time and Size of Solution Candidate Set Variable Per PLSA Parameter (VTS)**

There are two heating schemes possible when the number of iterations  $t_i$  per PLSA parameter is a constant value  $t_i = t_{\max}/n$ . One scheme we call VTS-S, in which the next solution candidate set is filled with new solution candidates until, for  $N_{\text{stag}}$  solutions, no improvement in fitness is observed. In this case the same procedure is applied to the next iteration using the same parameter  $p_i$ . If  $t_i$  iterations have been performed for  $p_i$ , the next PLSA parameter  $p_{i+1}$  is taken.

In the other heating scheme, which we call VTS-T, the filling of the next solution candidate set is stopped if, for  $T_{\text{stag}}$  seconds, the quality of the best solution in the solution candidate set has stagnated (i.e. has not improved).

### **Number of Iterations and Amount of Time Variable Per PLSA Parameter (VIT)**

Here again there are two possible variations. The first, called VIT-I, considers the number of iterations as the time constraint. The next PLSA parameter value is taken when for a number  $t_{\text{stag}}$  of iterations the quality of the best solution in the solution candidate set has not improved. As a consequence, for each parameter a different amount of time may be considered until the stagnation condition is fulfilled.

The alternative VIT-T is to define the time constraint in seconds. In this case, the next PLSA parameter value is taken when, for  $T_{\text{stag}}$  seconds, no improvement in fitness was achieved. As a consequence, for each parameter a different number of iterations may be considered until the stagnation condition is fulfilled.

In the next chapter we will describe some experiments to verify the simulated heating technique.



## **Chapter 9**

### **Simulated Heating Experiments**

Hybrid global/local search techniques are most effective in problems with complicated search spaces, and problems for which local search techniques have been developed that make maximum use of problem-specific information. We investigate the effectiveness of the simulated heating approach on the voltage scaling problem for embedded multiprocessors described in Section 4.3, as well as a memory compaction problem in embedded systems. These problems are very different in structure, but both have vast and complicated solution spaces. In addition, the parameterized local search algorithms (PLSA) for these applications exhibit a wide range of accuracy/complexity trade-offs. To further illustrate the utility of simulated heating, we demonstrate its use on the well-known binary knapsack problem.

#### **9.1 Simulated Heating for Voltage Scaling**

The problem of dynamic voltage scaling for multiprocessors was introduced in Section 4.3 and two different PLSAs for the problem were presented in Section 8.1.1. In this section we explain how we used simulated heating to solve this problem. Experimental results are given in Section 9.4.

### 9.1.1 Voltage Scaling Problem Statement

We assume that a schedule has been computed beforehand so that the ordering of the tasks on the processors is known. The optimization problem we address consists of finding the voltage vector  $V = (\nu_1, \nu_2, \dots, \nu_n)$  for the  $n$  tasks in the application graph, such that the energy per computation period (average power) is minimized and the throughput satisfies some pre-specified constraint (e.g., as determined by the sample period in a DSP application). For each task, as its voltage is decreased, its energy is decreased and its execution time is increased, as described in [9]. The computation period is determined from the period graph. A simple example is shown in Figure 9.1. Here we can see that by decreasing the voltage on task  $B$ , the average power is reduced while the execution time is unchanged. There is a potentially vast search space for many practical applications. For example, if we consider discrete voltage steps of 0.1 Volts over a range of 5 Volts, there are  $n^{50}$  possible voltage vectors  $V$  from which to search. The number of tasks  $n$  in an application may be in the hundreds.

### 9.1.2 GSA: Evolutionary Algorithm for Voltage Scaling

Each solution  $s$  is encoded by a vector of positive real numbers of size  $N$  representing the voltage assigned to each of the  $N$  tasks in the application. The one-point crossover operator randomly selects a crossover point within a vector then interchanges the two parent vectors at this point to produce two new offspring. The mutation operator randomly changes one of the elements of the vectors to a new (positive) value. At each generation of the EA an entirely new population is created based on the crossover and mutation operators. The crossover probability was 0.9, the mutation probability was 0.1, and the population size was 50.

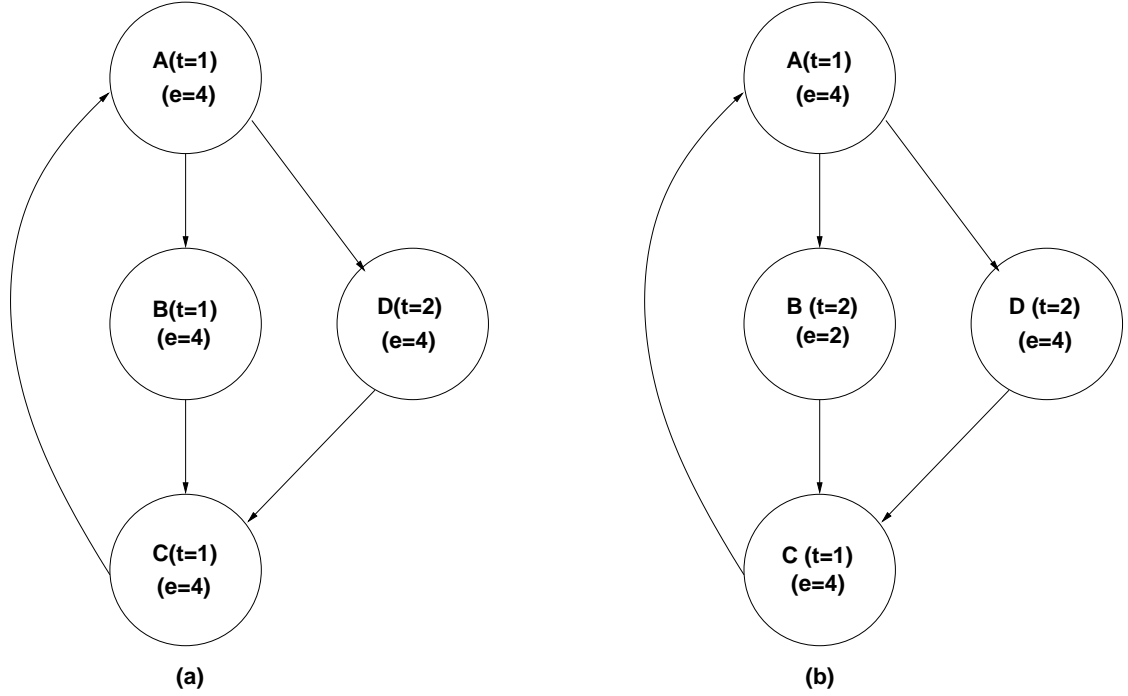


Figure 9.1: **(a)** Period Graph before voltage scaling. The numbers represent execution times ( $t$ ) and energies ( $e$ ) of the tasks. The execution period is determined by the longest cycle,  $A \rightarrow B \rightarrow C$ , whose sum of execution times is 4 units. The energy of each task is 4 units. the average power is 4 units (16 total energy divided by period of 4).

**(b)** After voltage scaling. The voltage on task  $B$  has been reduced, increasing its execution time from 1 unit to 2 units and decreasing its energy consumption from 4 units to 2 units. The overall execution period is still 4 units since both cycles  $A \rightarrow D \rightarrow C$  and  $A \rightarrow B \rightarrow C$  now have execution time of 4. The average power is 3.5 units (14 total energy divided by period of 4).

## 9.2 Simulated Heating for Memory Cost Minimization

In order to further demonstrate the simulated heating technique, we apply it to another optimization problem in electronic design automation that has a complicated search space. This section will explain both the PLSA and the GSA for this problem. Experimental results are given in Section 9.4.

### 9.2.1 Background

Digital signal processing (DSP) applications can be specified as dataflow graphs [17]. As explained in Chapter 2, in dataflow a computational specification is represented as a directed graph in which vertices (*actors*) specify computational functions of arbitrary complexity, and edges specify FIFO communication between functions. A *schedule* for a dataflow graph is simply a specification of the order in which the functions should execute. A given DSP application can be accomplished with a variety of different schedules—we would like to find a schedule which minimizes the memory requirement. A *periodic schedule* for a dataflow graph is a schedule that invokes each actor at least once and produces no net change in the number of data items queued on each edge. A software synthesis tool generates application programs from a given schedule by piecing together (*inlining*) code modules from a predefined library of software building blocks associated with each actor. The sequence of code modules and subroutine calls that is generated from a dataflow graph is processed by a buffer management phase that inserts the necessary target program statements to route data appropriately between actors.

The scheduling phase has a large impact on the memory requirement of the final implementations, and it is this memory requirement we wish to minimize in our optimization. The key components of this memory requirement are the code size cost (the

sum of the code sizes of all inlined modules, and of all inter-actor data transfers). Even for a simple dataflow graph, the underlying range of trade-offs may be very complex. We denote a *schedule loop* with the notation  $(nT_1T_2 \dots T_m)$ , which specifies the successive repetition  $n$  times of a subschedule  $T_1T_2 \dots T_m$ , where the  $T_i$  are actors. A schedule that contains zero or more schedule loops is called a *looped schedule*, and a schedule that contains exactly zero schedule loops is called a *flat schedule* (thus, a flat schedule is a looped schedule, but not vice-versa).

Consider two schedules  $S_1 = (8YZ)(2YZ)$  and  $S_2 = X(10YZ)$  which repeat for the actors  $X$ ,  $Y$ , and  $Z$  the same number of times (1, 10, 10, respectively). The *code size* for schedules  $S_1$  and  $S_2$  can be expressed, respectively, as  $\kappa(X) + \kappa(Y) + \kappa(Z) + L_c$ , where  $L_c$  denotes the processor-dependent, code size overhead of a software looping construct, and  $\kappa(A)$  denotes the program memory cost of the library code module for an actor  $A$ . The code size of schedule  $S_1$  is larger because it contains more “actor appearances” than schedule  $S_2$  (e.g., an actor  $Y$  appears twice in  $S_1$  vs. only once in  $S_2$ ), and  $S_1$  also contains more schedule loops (2 vs. 1). The *buffering cost* of a schedule is computed as the sum over all edges  $e$  of the maximum number of buffered (produced, but not yet consumed) tokens that coexist on  $e$  throughout execution of the schedule. Thus, the buffering costs of  $S_1$  and  $S_2$  are 11 and 19, respectively. The *memory cost* of a schedule is the sum of its code size and buffering costs. Thus, depending on the relative magnitudes of  $\kappa(X)$ ,  $\kappa(Y)$ ,  $\kappa(Z)$ , and  $L_c$ , either  $S_1$  or  $S_2$  may have lower memory cost.

### 9.2.2 MCMP Problem Statement

The *memory cost minimization problem (MCMP)* is the problem of computing a looped schedule that minimizes the memory cost for a given dataflow graph, and a given set of actor and loop code sizes. It has been shown that this problem is NP-complete [17]. A

tractable algorithm called *CDPPO* (code size dynamic programming post optimization), which can be used as a local search for MCMP, has also been described [16, 111, 112]. In this work the CDPPO was applied uniformly at “full strength” (maximum accuracy/maximum run-time), and as conventionally done with local search techniques, did not explore application of its PLSA form. As explained below, the CDPPO algorithm can be formulated naturally as a PLSA with a single parameter such that accuracy and run-time both increase *monotonically* with the parameter value.

### 9.2.3 Implementation Details for MCMP

To solve the MCMP we use a GSA/PLSA hybrid where an evolutionary algorithm is the GSA and CDPPO is the PLSA. The evolutionary algorithm and parameterized CDPPO are explained below.

### 9.2.4 GSA: Evolutionary Algorithm for MCMP

Each solution  $s$  is encoded by an integer vector, which represents the corresponding schedule, i.e., the order of actor executions (*firings*). The decoding process that takes place in the local search/evaluation phase (step 5 in Figure 8.6) is as follows:

- First a repair procedure is invoked, which transforms the encoded actor firing sequence into a valid flat schedule.
- Next the parameterized CDPPO is applied to the resulting flat schedule in order to compute a (sub)optimal looping, and afterward the data requirement (buffering cost)  $D(s)$  and the program requirement (code size cost)  $P(s)$  of the software implementation represented by the looped schedule are calculated based on a certain processor model.

Finally, both  $D(s)$  and  $P(s)$  are normalized (the minimum values  $D_{\min}$  and  $P_{\min}$  and maximum values  $D_{\max}$  and  $P_{\max}$  for the distinct objectives can be determined beforehand) and a fitness is assigned to the solution  $s$  according to the following formula:

$$F(s) = 0.5 \frac{D(s) - D_{\min}}{D_{\max} - D_{\min}} + 0.5 \frac{P(s) - P_{\min}}{P_{\max} - P_{\min}} \quad (9.1)$$

Note that the fitness values are to be minimized here.

### 9.2.5 PLSA: Parameterized CDPPO for MCMP

The “unparameterized” CDPPO algorithm was first proposed in [16]. CDPPO computes an optimal parenthesization in a bottom-up fashion, which is analogous to dynamic programming techniques for matrix-chain multiplication [28]. Given a dataflow graph  $G = (V, E)$  and an actor invocation sequence (flat sequence)  $f_1, f_2, \dots, f_n$ , where each  $f_i \in V$ , CDPPO first examines all 2-invocation *sub-chains*  $(f_1, f_2), (f_2, f_3), \dots, (f_{n-1}, f_n)$  to determine an optimally-compact looping structure (*subschedule*) for each of these sub-chains. For a 2-invocation sub-chain  $(f_i, f_{i+1})$ , the most compact subschedule is easily determined: if  $f_i = f_{i+1}$ , then  $(2f_i)$  is the most compact subschedule, otherwise the original (unmodified) subschedule  $f_i f_{i+1}$  is the most compact. After the optimal 2-node subschedules are computed in this manner, these subschedules are used to determine optimal 3-node subschedules (optimal looping structures for subschedules of the form  $f_i, f_{i+1}, f_{i+2}$ ); and the 2- and 3-node subschedules are then used to determine optimal 4-node subschedules, and so on until the  $n$ -node optimal subschedule is computed, which gives a minimum code size implementation of the input invocation sequence  $f_1, f_2, \dots, f_n$ .

Due to its high complexity, CDPPO can require significant computational resources for a single application—e.g., we have commonly observed run-times on the order of

30-40 seconds for practical applications. In the context of global search techniques, such performance can greatly limit the number of neighborhoods (flat schedules) in the search space that are sampled. To address this limitation, however, a simple and effective parameterization emerges: we simply set a threshold  $M$  on the maximum sub-chain (subschedule) size to which optimization is attempted. This threshold becomes the parameter of the resulting *parameterized CDPPO* (PCDPPO) algorithm.

In summary, PCDPPO is a parameterized adaptation of CDPPO for addressing the schedule looping problem. The run-time and accuracy of PCDPPO are both monotonically nondecreasing functions of the algorithm “threshold” parameter  $M$ . In the context of the memory minimization problem, PCDPPO is a genuine PLSA.

### 9.3 Simulated Heating for Binary Knapsack Problem

In order to further illuminate simulated heating, we begin by demonstrating the technique on a widely known problem, namely the binary (0-1) knapsack problem (KP). This problem has been studied extensively, and good exact solution methods for it have been developed (e.g. see [88]). The exact solutions are based on either branch-and-bound or dynamic programming techniques. In this problem, we are given a set of  $n$  items, each with profit  $\Delta_j$  and weight  $w_j$ , which must be packed in a knapsack with weight capacity  $c$ . The problem consists of selecting a subset of the  $n$  items whose total weight does not exceed  $c$  and whose total profit is a maximum. This can be expressed formally as:

$$\text{maximize } z = \sum_{j=1}^n \Delta_j x_j \tag{9.2}$$

subject to

$$\sum_{j=1}^n w_j x_j \leq c \tag{9.3}$$



$$x_j \in 0, 1, j \in 1, \dots, n \quad (9.4)$$

where  $x_j = 1$  if item  $j$  is selected, and  $x_j = 0$  otherwise.

Balas and Zemel [4] first introduced the “core problem” as an efficient way of solving KP, and most of the exact algorithms have been based on this idea. Pisinger [87] has modeled the hardness of the core problem and noted that it is important to test at a variety of weight capacities. He proposed a series of randomly generated test instances for KP. In our experiments we generate test instances using the test generator function described in appendix B of [87]. We compare our results to the exact solution described in [88], for which the C-code can be found in [110].

### 9.3.1 Implementation

To solve the KP we use a GSA/PLSA hybrid as discussed in Section 8.3 where an evolutionary algorithm is the global search algorithm (GSA) and a simple pairwise exchange is the parameterized local search algorithm (PLSA). The evolutionary algorithm and local search are explained below:

#### GSA: Evolutionary Algorithm

Each candidate solution  $s$  is encoded as a binary vector  $\vec{x}$ , where  $x_j$  are the binary decision variables from equation 9.4 above. The weight of a given solution candidate  $s$  is  $w_s = \sum_{j=1}^n x_j w_j$ , and the profit of  $s$  is  $\Delta_s = \sum_{j=1}^n x_j \Delta_j$ . The sum of the profits of all items is defined as  $\Delta_t = \sum_{j=1}^n \Delta_j$ . We define a fitness function which we would like to *minimize*:

$$F(s) = \begin{cases} \Delta_t - \Delta_s & \text{if } w_s \leq c \\ \Delta_t + w_s & \text{if } w_s > c \end{cases} \quad (9.5)$$

Thus we penalize solution candidates whose weight exceeds the capacity, and seek

to maximize the profit. The  $\Delta_t$  term was added so that  $F(s)$  is never negative. For the KP experiments we used a standard simple genetic algorithm described in [43] with one point crossover, crossover probability 0.9, non-overlapping populations of size popsize = 100, and elitism.

### **Parameterized Local Search for Knapsack Problem**

At the beginning of the optimization algorithm, the items are sorted by increasing profit, so that  $\Delta_i \leq \Delta_j$  for all  $i < j$ . Given an input solution candidate  $s$ , the local search first computes its weight  $w_s$ . If  $w_s > c$ , items are removed ( $x_i$  set to zero) starting at  $i = 0$  until  $w_s \leq c$ . For local search parameter  $p = 1$ , this is the only operation performed. For  $p > 1$ , pair swap operations are also performed as explained in Figure 9.2, where we attempt to replace an item from the solution candidate with a more profitable item not included in the solution candidate. The number of such pair swap operations is  $p$ . Thus the local search algorithm requires more computation time and searches the local area more thoroughly for higher  $p$ . These are the monotonicity requirements expressed in Equation 8.1. We define parameter  $p = 0$  as no local search—i.e. the optimization is an evolutionary algorithm only, and no local search is performed.

## **9.4 Experiments**

In this section we present experiments designed to examine several aspects of simulated heating for the two embedded systems applications. We would like to know how simulated heating compares to the standard hybrid technique of using a fixed parameter (fixed  $p$ ). We summarize the fixed  $p$  results for all problems for different values of  $p$ . We examine how the optimal value of  $p$  for the standard hybrid method depends on the application.

$$s_{\text{in}}, F, s_{\text{out}}$$

**input:** fitness function  $F$

$$i \leftarrow n$$
$$\text{bestScore} \leftarrow F(s_{\text{in}})$$
$$\mathbf{while} \ (i > 0) \wedge (\text{count} < pn)$$

```

do {
  j = -1
  i = i - 1
  while (j < i) ∧ (count < pn)
  do {
    j = j + 1
    if sout[i] ≠ sout[j]
    then {
      temp = sout[i]
      sout[i] = sout[j]
      sout[j] = temp
      score = F(sout)
      if score < bestScore
      then {
        bestScore = score
      }
      else {
        temp = sout[i]
        sout[i] = sout[j]
        sout[j] = temp
      }
    }
  }
}

```

Figure 9.2: Pseudo-code for pair swap local search for binary knapsack problem.

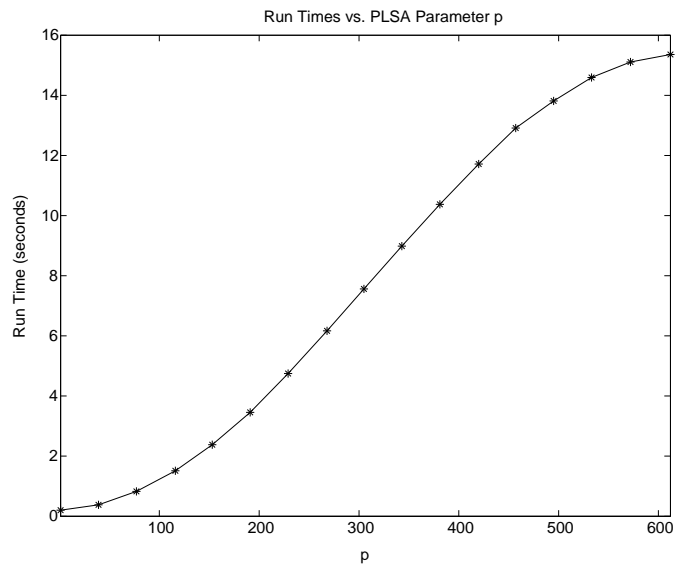
Next we compare both the static and dynamic heating schemes to the standard approach, and to each other. For the static heating experiments, we utilize the FIS and FTS strategies. Recall that **FIS** refers to **fixed** number of **iterations** and population **size** per parameter, and **FTS** refers to **fixed time** and population **size** per parameter. For the dynamic heating experiments, we utilize the two variants of the **VIT** strategy (**variable iterations** and **time** per parameter). We also examine the role of parameter range and population size on the optimization results.

#### 9.4.1 PLSA Run-Time and Accuracy for Voltage Scaling and MCMP

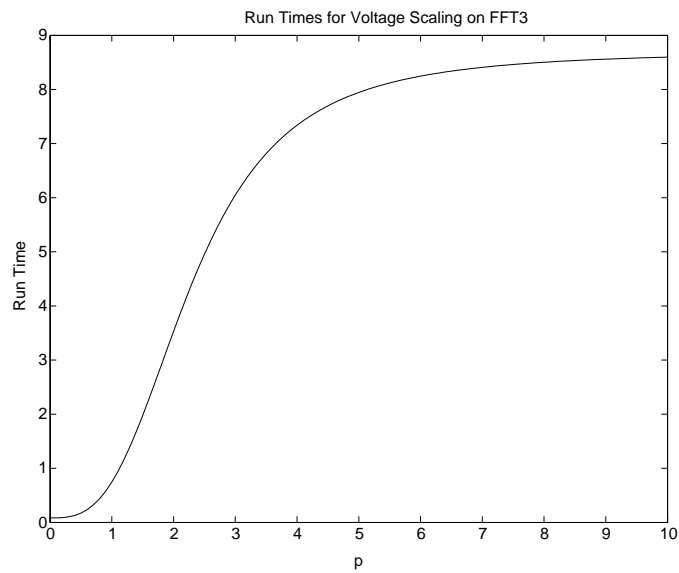
Recall that there is a trade-off between accuracy and run-time for the PLSA. Lower values of local search parameter  $p$  mean the local search executes faster, but is not as accurate. Figure 9.3 shows how the run-time of the PLSA varies with  $p$  for the two applications. It can be seen that the monotonicity property, Equation 8.1, is satisfied for the PLSAs.

#### 9.4.2 Standard Hybrid Approach for Voltage Scaling and MCMP

The standard approach to hybrid global/local searches is to run the local search at a fixed parameter. We present results for this method below. It is important to note that, for a fixed optimization run-time, the optimal value of local search parameter  $p$  can depend on the run-time and data input and cannot be predicted in advance. Figure 9.4 shows results for the MCMP optimization using fixed values of  $p$  (standard approach—no heating), for 11 different initial populations, for population sizes  $N = 100$  and  $N = 200$ . The y-axis on these graphs corresponds to the memory cost of the optimized schedule so that lower values are better. The x-axis corresponds to the fixed  $p$  value. For each value of  $p$ , the hybrid search was run for a time budget of 5 hours with a fixed value of  $p$ .

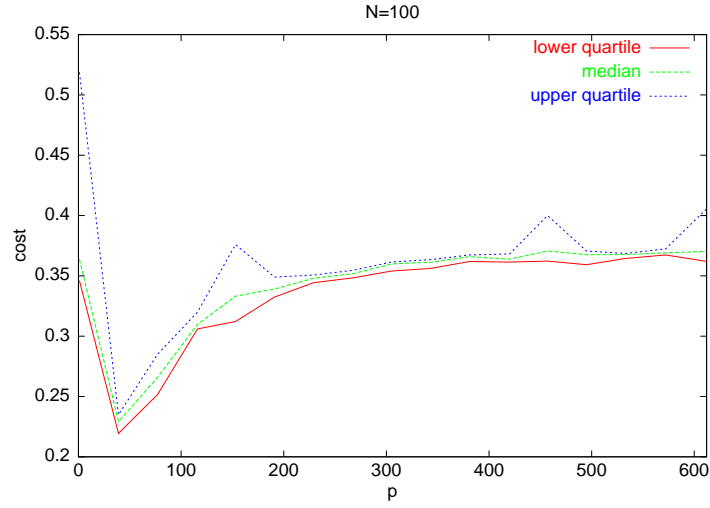


(a) MCMP application.

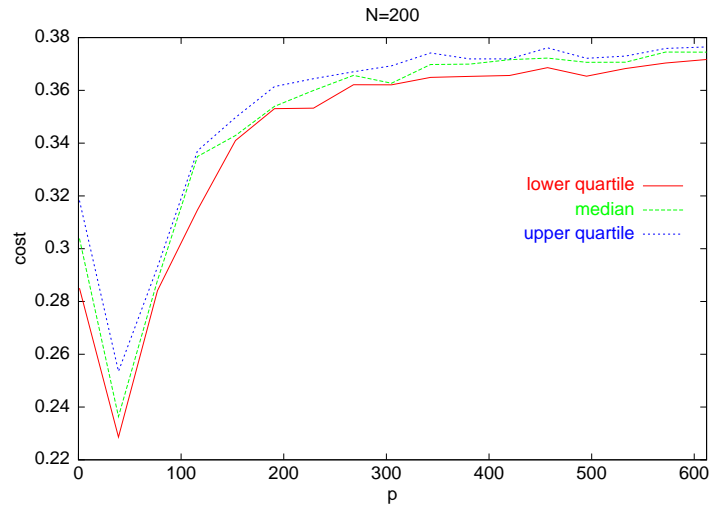


(b) Voltage scaling application.

Figure 9.3: Local search run times vs.  $p$  for MCMP application (a) and voltage scaling application (b).



(a)  $N = 100$



(b)  $N = 200$

Figure 9.4: Standard hybrid approach to MCMP application using fixed PLSA parameter  $p$ . Hybrid was run for 5 hours at each value of  $p$ . Population size for GA was  $N = 100$  in 9.4(a) and  $N = 200$  in 9.4(b). Median, lower quartile, and upper quartile of 11 different runs shown in the three curves for each  $p$ . (Lower memory cost is better).

The same set of initial populations was used. From these graphs, it can be seen that the local search performs best for values of  $p$  around 39. Figure 9.5 shows the number of iterations (generations in the GSA) performed for each value of  $p$ . As  $p$  increases, fewer generations can be completed in the fixed optimization run time.

Figure 9.6 shows results for the voltage scaling application on 6 different input dataflow graphs, for fixed values of  $p$  (no heating), for 11 different initial populations, using both hill climb and Monte Carlo local search methods. For each value of  $p$ , the hybrid search was run for a time budget of 20 minutes with a fixed value of  $p$ . The y-axis on the graph corresponds to the ratio of the optimized average power to the initial power, so that lower values are better. For each  $p$ , the same set of initial populations was used. From these graphs, it can be seen that the best value of  $p$  may also depend on the specific problem instance.

### 9.4.3 Static Heating Schemes for Voltage Scaling and MCMP

For the MCMP application, the run-time limit for the hybrid was set to  $T_{\max} = 5$  hours. Two sets of PLSA parameters were used,  $R^1 = [1, 153, 305, 457, 612]$  and  $R^2 = [1, 39, 77, 116, 153]$ . The value of  $p = 612$  corresponds to the total number of actor invocations in the schedule for the MCMP application and is thus the maximum (highest accuracy) possible. The parameter set  $R^2$  was chosen so that it is centered around the best fixed  $p$  values. Figure 9.7 summarizes the results for the MCMP application with GSA population size  $N = 100$ . In Figure 9.7, eleven runs were performed for each heating scheme and for each parameter set. The box plot <sup>1</sup> Figure 9.7(a) corresponds

---

<sup>1</sup>The ‘box’ in the box plot stretches from the 25<sup>th</sup> percentile (‘lower hinge’) to the 75<sup>th</sup> percentile (‘upper hinge’). The median is shown as a line across the box. The ‘whisker’ lines are drawn at the 10<sup>th</sup> and 90<sup>th</sup> percentiles. Outliers are shown with a ‘+’ character.

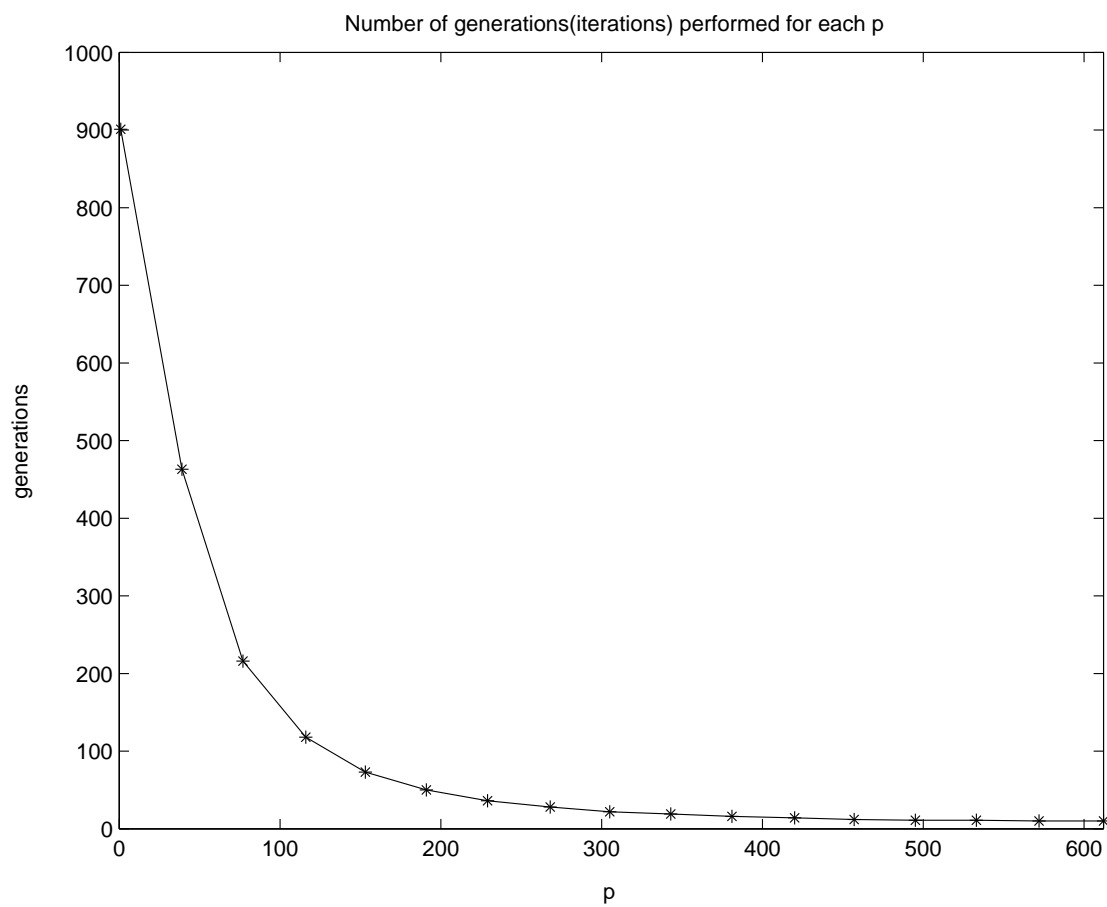
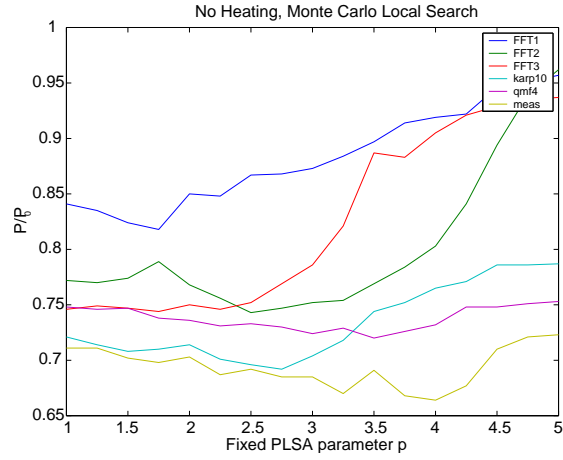
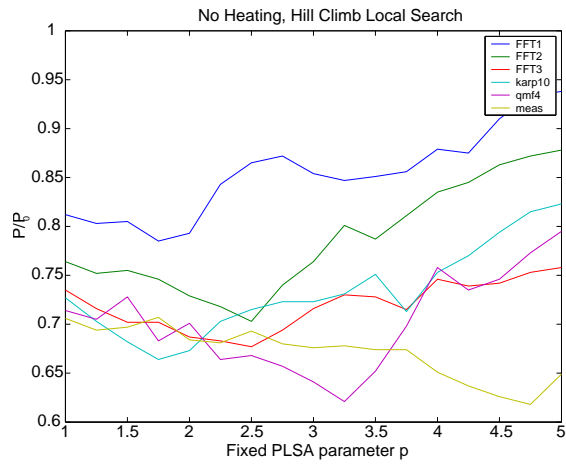


Figure 9.5: Standard hybrid approach (fixed  $p$ , no heating), MCMP application, using a fixed run time. Number of generations completed is shown for hybrids utilizing different values of  $p$ . Fewer generations are completed for higher  $p$ .





(a) Monte Carlo local search.



(b) Hill climb local search.

Figure 9.6: Standard hybrid approach using fixed PLSA parameters, voltage scaling application, with Monte Carlo local search in 9.6(a) and hill climb local search in 9.6(b). Hybrid was run for 20 minutes at each value of  $p$ . Median of 11 runs for each  $p$ . Lower values of power are better. We see that the optimal value of  $p$  is different for the six different input dataflow graphs.

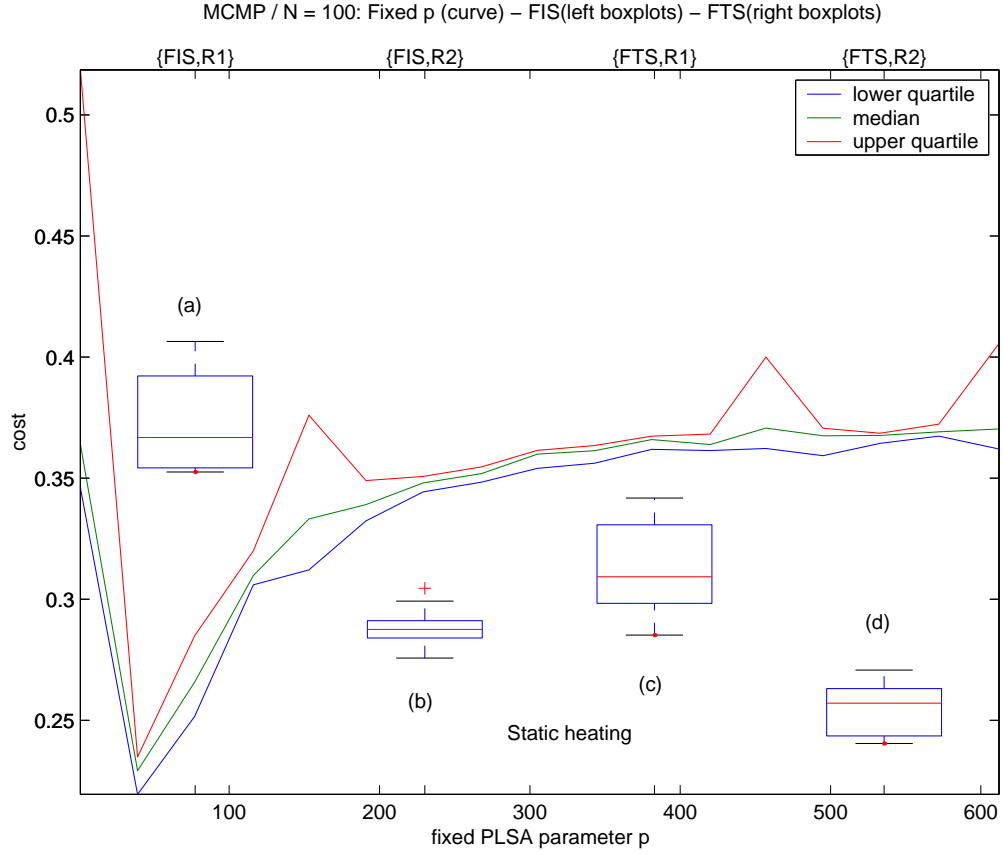


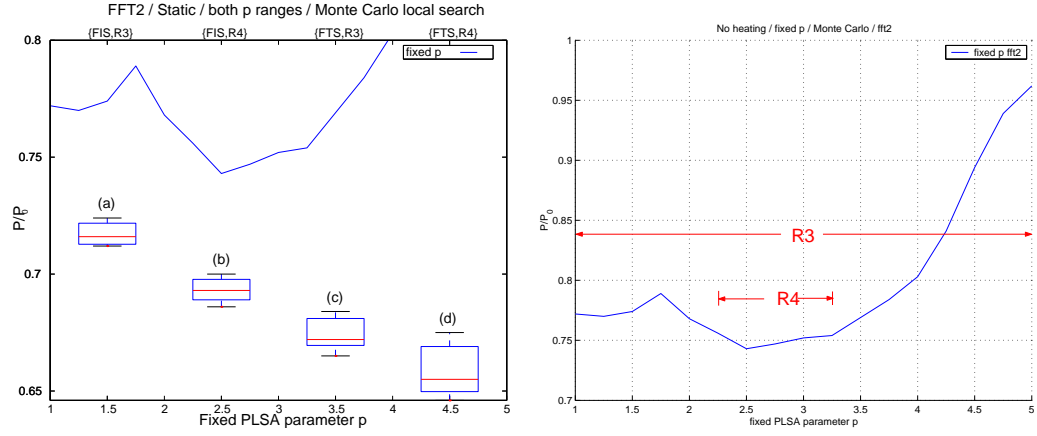
Figure 9.7: Static heating for MCMP with the local search parameter  $p$  varied in two different ranges—the first range covers all possible values ( $1 - 612$ ), while the second range ( $1 - 153$ ) is concentrated around the best fixed  $p$  value. (a)[FIS,  $R^1$ ], (b)[FIS,  $R^2$ ], (c)[FTS,  $R^1$ ], (d)[FTS,  $R^2$ ]. The solid curve depicts the standard hybrid approach for different values of  $p$ . Lower values of cost are better. The box plots display the static heating results. The solid line across the box represents the median over all calculations. The lowest cost is obtained for the standard hybrid approach with  $p = 39$ . The best static heating scheme is (d), corresponding to FTS operating in the restricted parameter range which includes  $p = 39$ . We note that this value of  $p$  could not be determined in advance, and could only be found by running the standard hybrid solution for all values of  $p$ .

Heating scheme		Iterations per parameter $p$							
type	range	1	39	77	115	153	305	457	612
FIS	[1,612]	4	x	x	x	4	4	4	4
FIS	[1,153]	33	33	33	33	33	x	x	x
FTS	[1,612]	176	x	x	x	14	4	2	2
FTS	[1,153]	175	94	42	23	14	x	x	x

Table 9.1: Iterations performed per parameter value for four different heating schemes for MCMP. The numbers correspond to a single optimization run. For the other ten runs they look slightly different.

to FIS with parameter set  $R^1$ . Figure 9.7(b) corresponds to FIS with parameter set  $R^2$ . Figure 9.7(c) corresponds to FTS with parameter set  $R^1$ . Figure 9.7(d) corresponds to FTS with parameter set  $R^2$ . The solid curves in Figure 9.7 are the results for fixed  $p$ . Table 9.1 summarizes the iterations performed for each parameter for both FIS and FTS with both parameter ranges.

For the voltage scaling application, we ran the static heating optimization for a runtime of  $T_{\max}$  minutes. For FIS and FTS, the parameter sets used were  $R^3 = [1, 2, 3, 4, 5]$  and  $R^4 = [2.25, 2.50, 2.75, 3.00, 3.25]$ . The parameter set  $R^3$  was chosen by examining the fidelity of the period graph estimator. Recall that the PLSA parameter  $p$  is related to the re-simulation threshold. It is observed that for  $p < 1$  the fidelity of the estimator is poor. For  $p$  greater than 5, with the voltage increments used, the re-simulation threshold is so small that simulation is done almost every time. This corresponds to the highest accuracy setting. The parameter set  $R^4$  was chosen to center around the best fixed  $p$  values. Results for FIS and FTS on the FFT2 application using the Monte Carlo local search are shown in Figure 9.8. The box plot in Figure 9.8(a) corresponds to FIS with



(a) Static heating(box plots),  $p$  fixed (upper curve).

(b) The 2 ranges for fixed  $p$ .

Figure 9.8: Static heating for voltage scaling with different parameter ranges— (a)[FIS,  $R^3$ ], (b)[FIS,  $R^4$ ], (c)[FTS,  $R^3$ ], (d)[FTS,  $R^4$ ] (shown in the four box plots) compared with the standard hybrid method results (fixed values of  $p$  shown in the solid line). Here the static heating schemes all perform better than the standard hybrid approach. The first parameter range includes all values of  $p$ , while the second range is centered around the best fixed  $p$  value. This is shown in more detail in 9.8(b).

parameter range  $R^3$ . Figure 9.8(b) corresponds to FIS with parameter range  $R^4$ . Figure 9.8(c) corresponds to FTS with parameter set  $R^3$ . Figure 9.8(d) corresponds to FTS with parameter range  $R^4$ . The solid curves in the figure are the results for fixed  $p$ .

#### 9.4.4 Dynamic Heating Schemes for Voltage Scaling and MCMP

We performed the dynamic heating schemes VIT.I and VIT.T for both the MCMP and voltage scaling applications. Recall that VIT stands for variable iterations and time per parameter; during the optimization the next PLSA parameter is taken when, for a given number  $t_{\text{stag}}$  of iterations (VIT.I) or a given time  $T_{\text{stag}}$  (VIT.T), the quality of the solution candidate has not improved.

For the MCMP application, the run-time limit for the hybrid was set to  $T_{\text{max}} = 5$  hours and the same two sets of PLSA parameters were used as in the static heating case. Eleven runs were performed for all cases. Results for dynamic heating on the MCMP application are shown in Figure 9.9 For the voltage scaling application, the run time was  $T_{\text{max}} = 20$  minutes. Results for voltage scaling with VIT.I and VIT.T using the Monte Carlo local search are shown in Figure 9.10. For the dynamic heating schemes, the search algorithm operates with a given PLSA parameter until the quality of the best solution has not improved for either  $t_{\text{stag}}$  iterations (VIT.I) or  $T_{\text{stag}}$  seconds (VIT.T). It is therefore interesting to observe the amount of time spent on each parameter during the optimization. This is illustrated in Figure 9.11.

#### 9.4.5 Knapsack PLSA Run-Time and Accuracy

To test the binary knapsack problem, we generated 1000 pseudo-random test instances for each technique as suggested in [87]. The weights and profits in these instances were strongly correlated. The weight capacity  $c_i$  of the  $i$ th instance is given by  $c_i =$

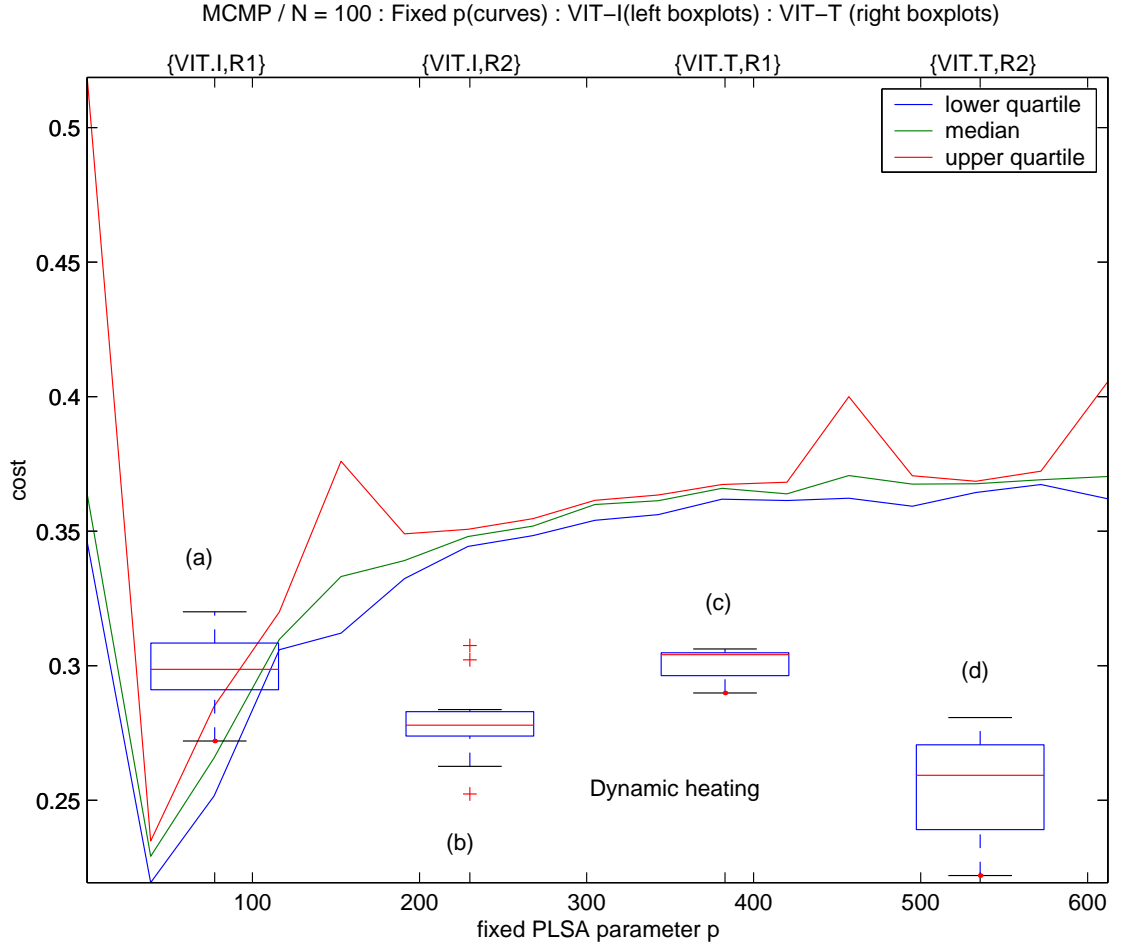


Figure 9.9: Dynamic heating for MCMP with different parameter ranges depicted by the four box plots—(a)[VIT.I, $R^1$ ], (b)[VIT.I, $R^2$ ], (c)[VIT.T, $R^1$ ], (d)[VIT.T, $R^2$ ]. The solid line represents the standard hybrid technique with  $p$  fixed at different values from 1 to 612. The solid lines across the boxes represents the median over all calculations. The lowest cost is obtained for the standard hybrid approach with  $p = 39$ . The best dynamic heating scheme is (d), corresponding to VIT.T operating in the restricted parameter range which includes  $p = 39$ . We note that this value of  $p$  could not be determined in advance, and could only be found by running the standard hybrid solution for all values of  $p$ .

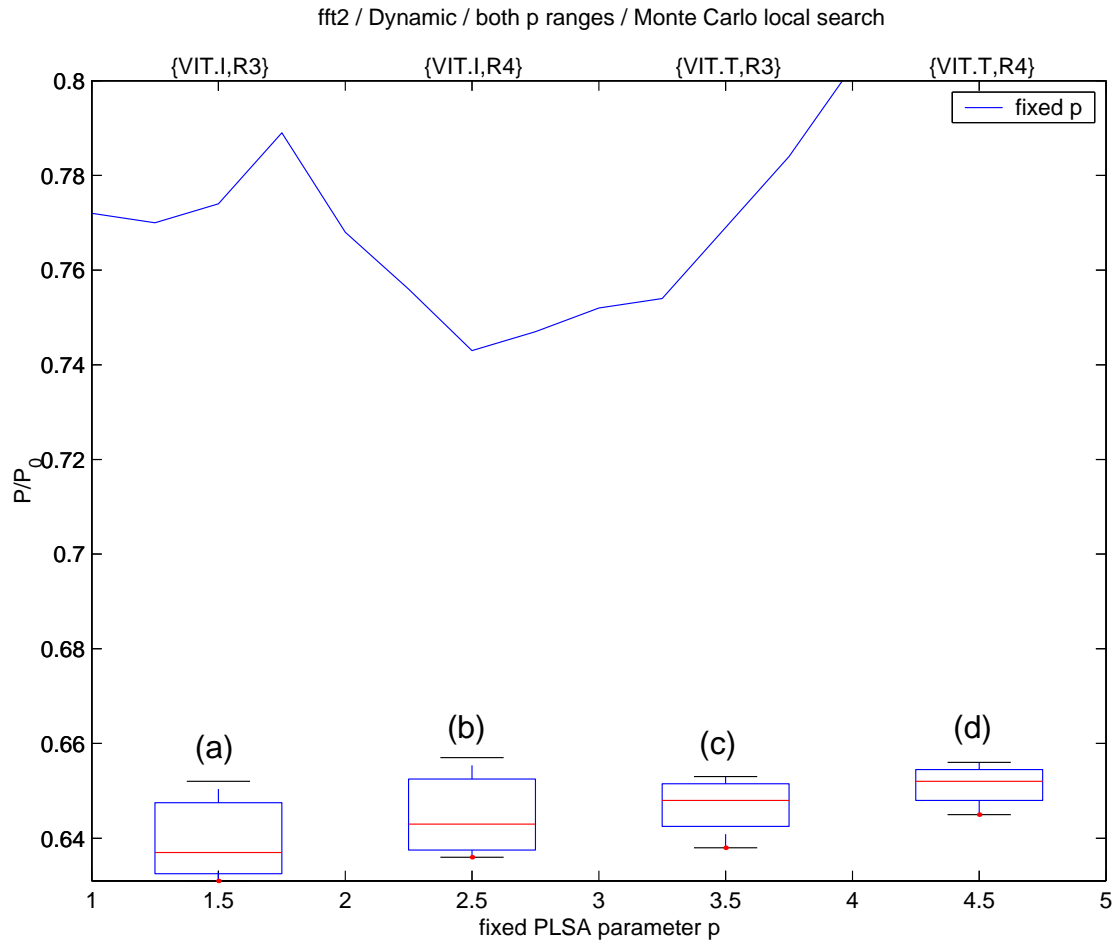
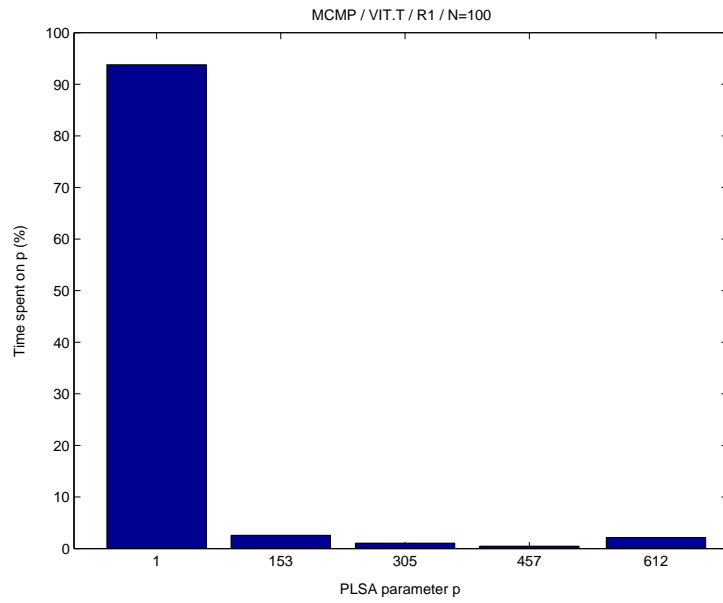
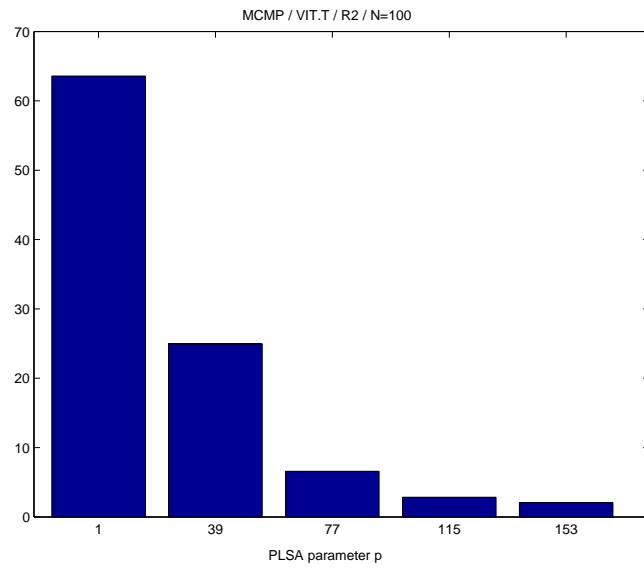


Figure 9.10: Dynamic heating for voltage scaling with different parameter ranges depicted by the four box plots—(a)[VIT.I,  $R^3$ ], (b)[VIT.I,  $R^4$ ], (c)[VIT.T,  $R^3$ ], (d)[VIT.T,  $R^4$ ]. VIT.T refers to variable iterations and time per parameter, with the next parameter taken if, for a given time, the solution has not improved. The solid curve depicts results for the standard hybrid approach. All the dynamic schemes outperform the standard hybrid (fixed  $p$ ) approach, with the lowest average power obtained for (a) VIT.I which utilizes the broader parameter range.



(a) Parameter range  $R^1$



(b) Parameter range  $R^2$

Figure 9.11: Percent of time spent on each parameter in range  $R^1$  (a) and in range  $R^2$  (b) for VIT.T.



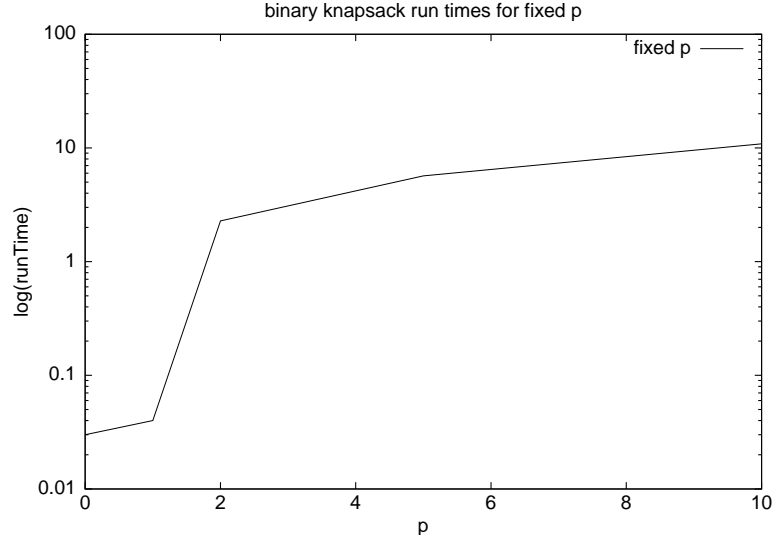


Figure 9.12: Local search run times vs.  $p$  for binary knapsack problem.

$\lfloor (iW)/1001 \rfloor$  where  $W$  is the sum of the weights of all items. For each test instance we compared the hybrid solution with an exact solution to the problem using the method given in [88]. We defined an error sum over all the problem instances as a figure of merit for the hybrid solution technique:

$$\epsilon = \sum_{i=1}^{1000} (\alpha_i - \beta_i) \quad (9.6)$$

where  $\alpha_i$  is the profit given by the exact solution and  $\beta_i$  is the profit given by the hybrid solution.

Figure 9.12 shows how the run-time of the pair swap PLSA increases with  $p$ . Figure 9.13 depicts the sum of errors (Equation 9.6) for the binary knapsack problem for different values of  $p$  with the number of generations fixed at 10. We can see that higher values of  $p$  produce smaller error, at the expense of increased run time. Thus the pair swap PLSA satisfies the monotonicity requirement from Equation 8.1.

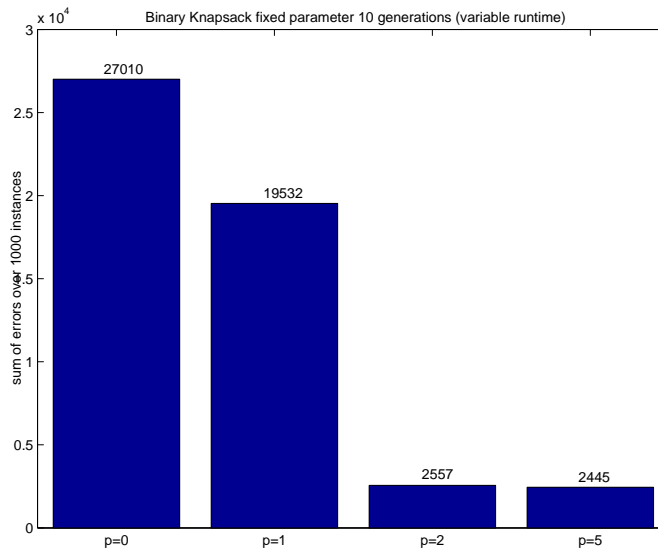


Figure 9.13: **Standard hybrid approach** for binary knapsack (fixed  $p$ , no heating) using a **fixed number of generations** and not fixing overall hybrid run time. **Cumulative error** shown for hybrids utilizing different  $p$ . Higher  $p$  is more accurate but requires longer run times.

#### 9.4.6 Knapsack Standard Hybrid Approach

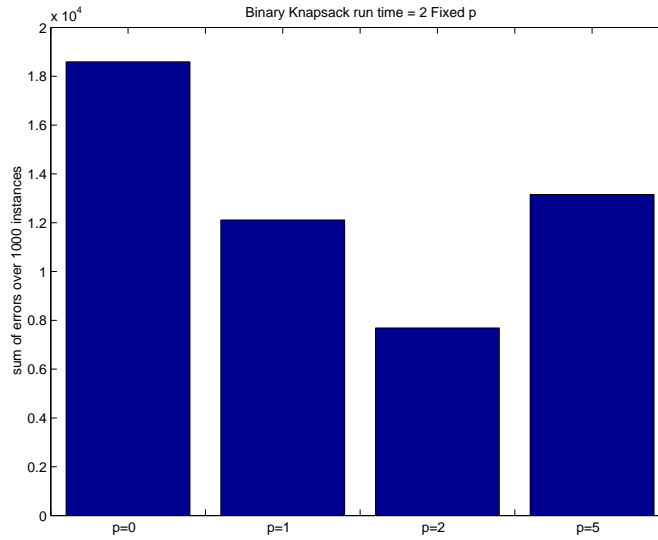
The standard hybrid approach to hybrid global/local searches is to run the local search at a fixed parameter. This is shown in Figure 9.14 for different values of  $p$  and for two different run times. Here the y-axis corresponds to the sum of errors over all test cases (Equation 9.6). We see that, for a fixed optimization run-time, the optimal value of local search parameter  $p$  using the standard hybrid approach can depend on the run-time and data input—for a run time of 2 seconds, the best value of  $p$  is 2, while for a run time of 5 seconds, the best value of  $p$  is 5. We note here and with the other applications studied that this value of  $p$  cannot be predicted in advance.

#### 9.4.7 Knapsack Static Heating Schemes

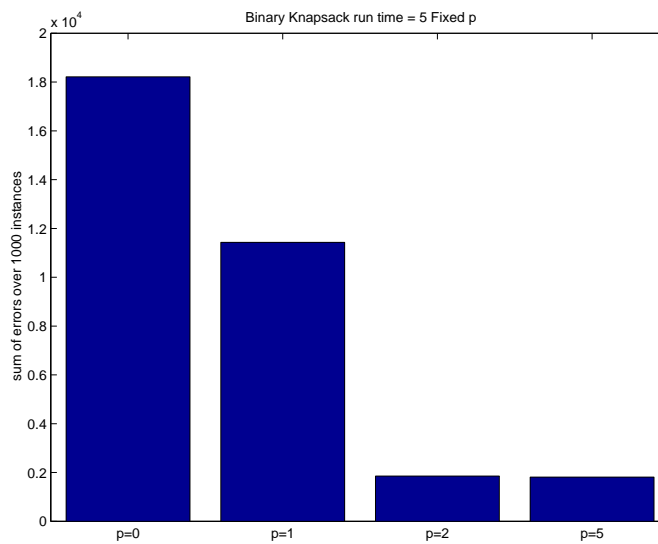
The static heating schemes FIS and FTS were performed for the binary knapsack problem. Results are shown in Figure 9.15 for run times of 1 and 5 seconds, and compared with the standard hybrid approach for different values of  $p$ . It can be seen that the static heating scheme outperformed the standard hybrid approach, and that this improvement is greater for the shorter run times.

#### 9.4.8 Knapsack Dynamic Heating Schemes

The dynamic heating schemes VIT.I and VIT.T were performed for the binary knapsack application. Recall that VIT stands for variable iterations and time per parameter; during the optimization the next PLSA parameter is taken when, for a given number of iterations (VIT.I) or a given time (VIT.T), the quality of the solution candidate has not improved. Figure 9.16 shows results for these dynamic schemes. Results for static heating schemes are shown on the right for comparison. We observe that the dynamic

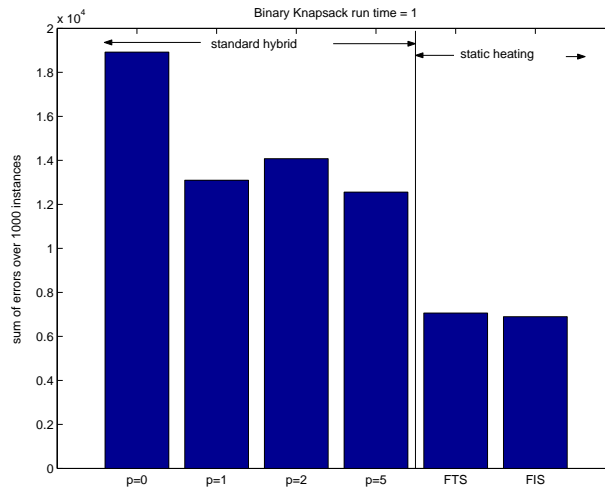


(a) Run time 2 seconds.

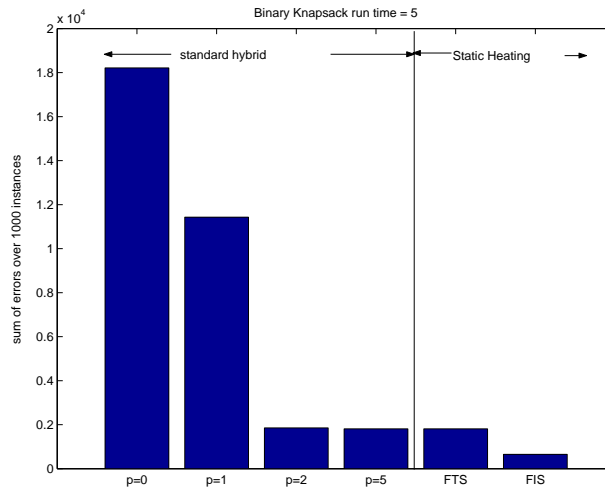


(b) Run time 5 seconds.

Figure 9.14: **Standard hybrid approach** applied to **binary knapsack** for different values of  $p$ , where  $p$  is fixed throughout. Y-axis is sum of errors. Run time is 2 seconds in (a) and 5 seconds in (b).

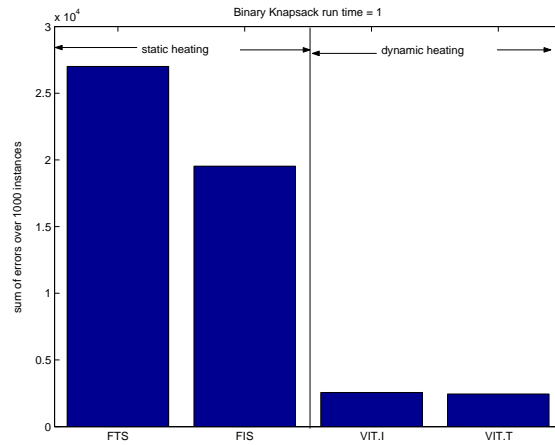


(a) Run time 1 second.

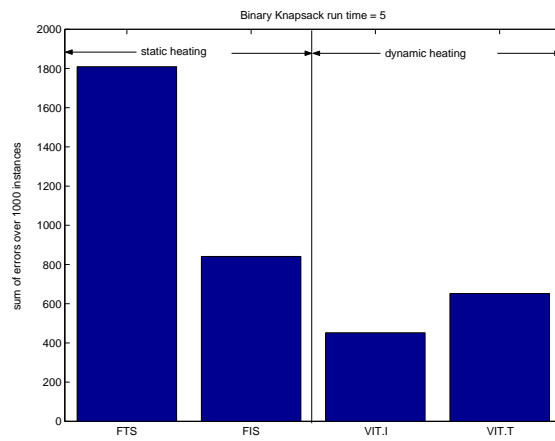


(b) Run time 5 seconds.

Figure 9.15: **Static heating** (2 bars on right) applied to **binary knapsack** compared to **the standard hybrid approach** (4 bars on left). Y-axis is sum of errors over all 1000 problem instances. The 4 bars on left correspond to the standard hybrid approach. Run time is 1 second in 9.15(a) and 5 seconds in 9.15(b).



(a) Run time 1 second



(b) Run time 5 seconds

Figure 9.16: **Dynamic heating for binary knapsack** (two bars on right) **compared to static heating** (two bars on left). VIT refers to variable iterations and time per parameter, with the next parameter taken if, for a given number of iterations (VIT.I) or a given time (VIT.T), the solution has not improved. Run time is 1 second in 9.16(a) and 5 seconds in 9.16(b). Y-axis is cumulative error over all problem instances (note the different y scales for the two plots).

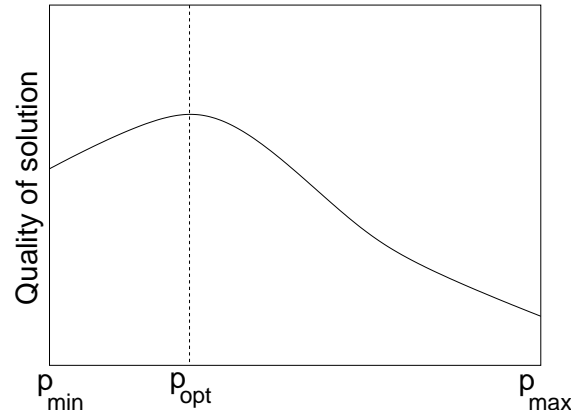


Figure 9.17: Relationship between the value of  $p$  and the outcome of the optimization process.

heating schemes outperform the static heating schemes significantly, and that the amount of improvement is greater for shorter run times.

## 9.5 Comparison of Heating Schemes

The results indicate that the choice of parameter  $p$  does affect the outcome of the optimization process. For the MCMP application, there is a pronounced region for fixed  $p$  values around  $p = 39$  where the hybrid (with  $p$  fixed) performs best. This is illustrated in Figure 9.4(a) (also shown as the solid curves in Figures 9.7 and 9.9). This is due to the trade-offs in accuracy and complexity with  $p$ . For smaller values of  $p$ , a larger number of iterations can be performed. (cf. Figure 9.5). It seems that there is a point beyond which increasing  $p$  decreases the performance of the hybrid algorithm. As illustrated in Figure 9.17, continuously increasing  $p$  starting from  $p = p_{\min}$  also increases the accuracy  $A(p)$  of the PLSA and therefore the effectiveness of the overall algorithm. However, when a certain runtime complexity  $C(p_{\text{opt}})$  of the PLSA is reached, the benefit

of higher accuracy may be outweighed by the disadvantage that the number of iterations that can be explored is smaller. As a consequence, values greater than  $p_{\text{opt}}$  may reduce the overall performance as the number of iterations is too low. Figure 9.6 depicts the performance of the hybrid with  $p$  fixed for the voltage scaling application on six different applications. It can be seen that the region of best performance is not as pronounced as in the MCMP application, and that this optimal value of  $p$  is different for different applications.

The observation that certain parameter ranges appear to be more promising than the entire range of permissible  $p$  values leads to the question of whether the heating schemes can do better when using the reduced range. One would expect that the static heating schemes, for which the number of iterations at each parameter is fixed beforehand, would benefit the most from the reduced range, since the hybrid would not be “forced” to run beyond  $p_{\text{opt}}$ . The dynamic heating schemes, by contrast, will continue to operate on a given parameter as long as the quality of the solution is improving. For the MCMP application, range  $R^2 = [1, 39, 77, 116, 153]$  is centered around the best fixed  $p$  values. Figures 9.7 through 9.10 compare the performance over the two parameter ranges. For the static heating optimizations in Figures 9.7 and 9.8, the performance is improved by using the reduced parameter ranges. The dynamic heating optimization in Figure 9.9 shows a smaller relative improvement. The dynamic heating optimization in Figure 9.10 actually shows a benefit to using the expanded parameter range. It is important to note that in practice one would not know about the characteristics of the different parameter ranges without first performing an optimization at each value. This would take much longer than the simulated heating optimization itself, so in practice the broader parameter range would probably be used. The data for fixed  $p$  for the MCMP problem (Figure 9.4(a) and 9.4(b)) demonstrate that it can be difficult to find the optimal



$p$  value and that this optimum may be isolated, i.e.  $p$  values close (e.g. 100) to optimum yield much worse results. If we calculate the median over all  $p$  values tried, the mean performance of the constant  $p$  approach is worse than the median performance of the FTS and VIT methods.

Figure 9.18 compares the results of the different heating schemes for the MCMP application with population size  $N = 100, 200, 50$  and parameter range  $R^1$ . Figure 9.19 compares the heating schemes for the voltage scaling application on different graphs for both types of local search.

Comparing the heating schemes across all different cases, we see that the dynamic heating schemes performed better in general than the static heating schemes. For all cases, the best heating scheme was dynamic. For the binary knapsack problem and the voltage scaling problem, simulated heating always outperformed the standard hybrid approach.

For the MCMP problem, there was one PLSA parameter where the standard hybrid approach slightly outperformed the dynamic, simulated heating approach. We note that in practice, one would need to scan the entire range of parameters to find this optimal value of fixed  $p$ , which is in fact equivalent to allotting much more time to this method. Thus, we can say that the simulated heating approach outperformed the standard hybrid approach in the cases we studied.

### 9.5.1 Effect of Population Size

Figure 9.20 shows the effect of the population size for MCMP for the static heating schemes. Figure 9.21 shows the effect of population size on the dynamic heating schemes for MCMP.

For FIS, smaller population sizes seem to be preferable. The larger number of itera-

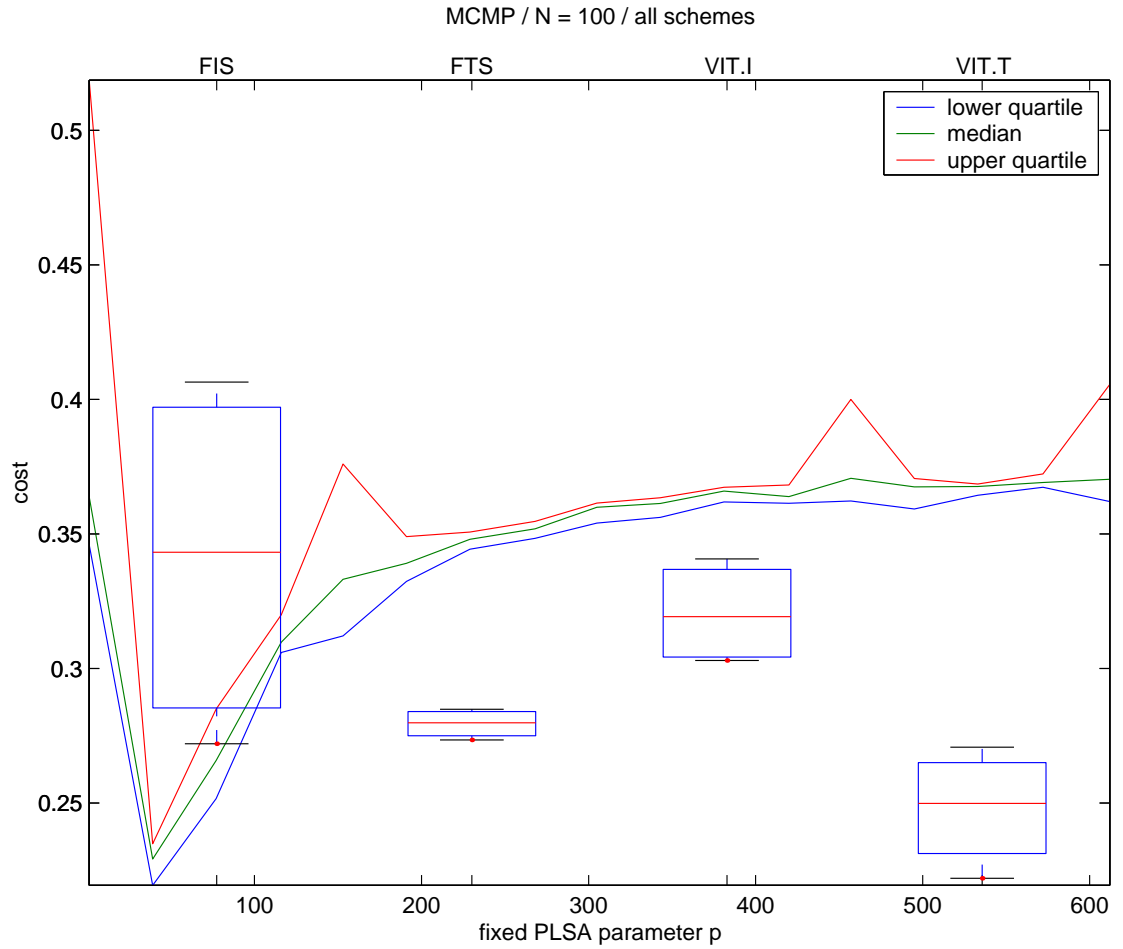
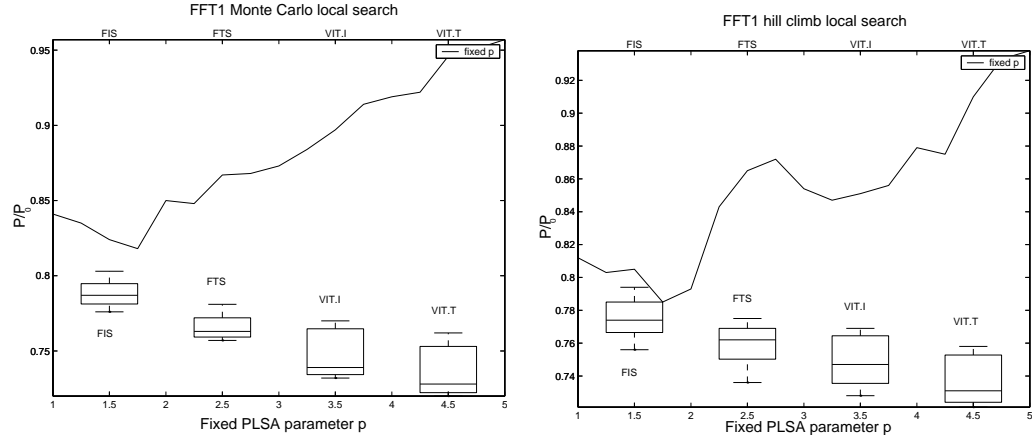


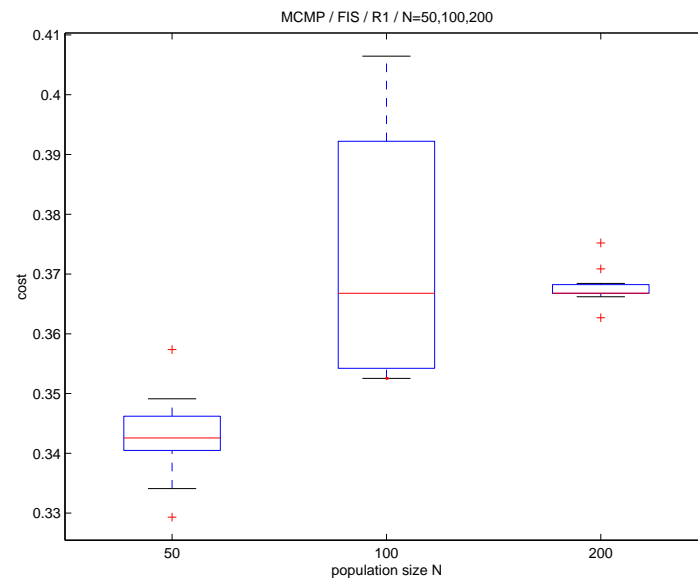
Figure 9.18: Comparison of heating schemes for MCMP with  $N = 100$ . The two box plots on left correspond to the static heating schemes. The two box plots on the right correspond to dynamic heating schemes. The best results (lowest memory cost) are obtained for the VIT.T dynamic heating scheme. This refers to variable iterations and time per parameter, where the parameter is incremented if the overall solution does not improve after a pre-determined time, called the stagnation time. The solid curve represents the standard hybrid approach applied at different values of fixed  $p$ . The point  $p = 39$  slightly outperforms the VIT.T scheme.



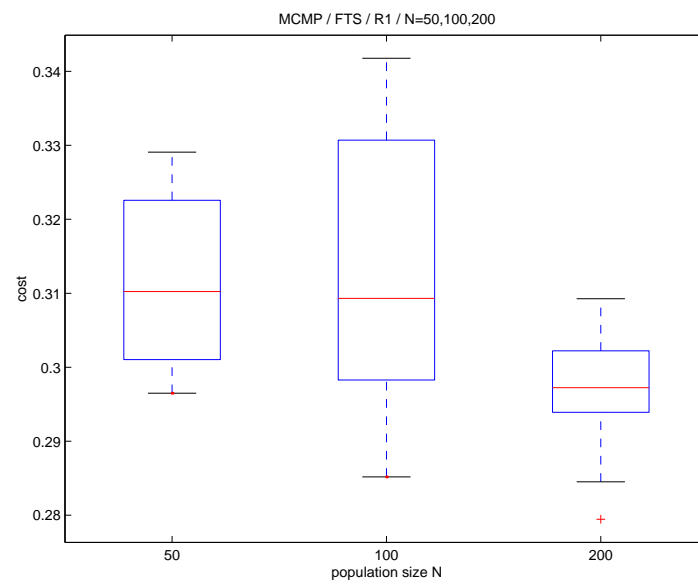
(a) Monte Carlo local search.

(b) Hill climb local search.

Figure 9.19: Comparison of heating schemes for voltage scaling with (a) Monte Carlo and (b) hill climb local search. The two box plots on left correspond to the FIS and FTS static heating schemes, while the two box plots on the right correspond to dynamic heating schemes VIT.I and VIT.T. The line across the middle of the boxes represents the median over the runs, while the ‘whisker lines’ are drawn at the 10th and 90th percentiles. The solid curve represents the standard hybrid approach applied at different values of fixed  $p$ . In this application, all the simulated heating schemes outperformed the standard hybrid approach. The best results were obtained for the dynamic VIT.T scheme.

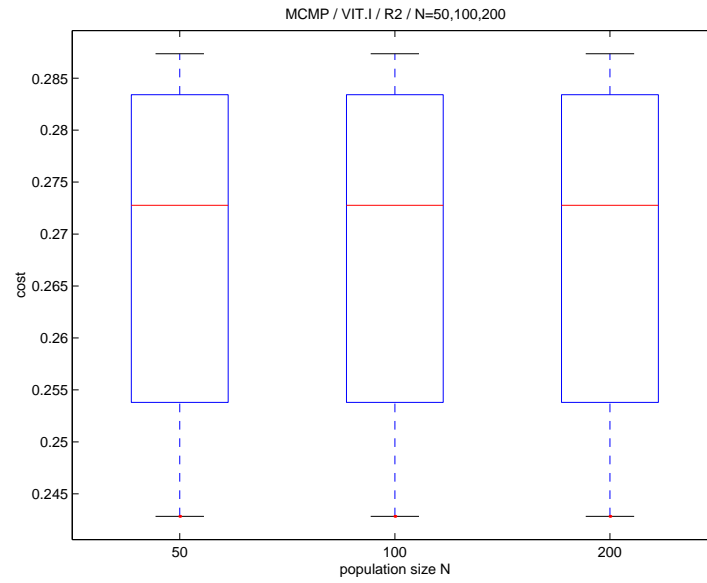


(a) FIS

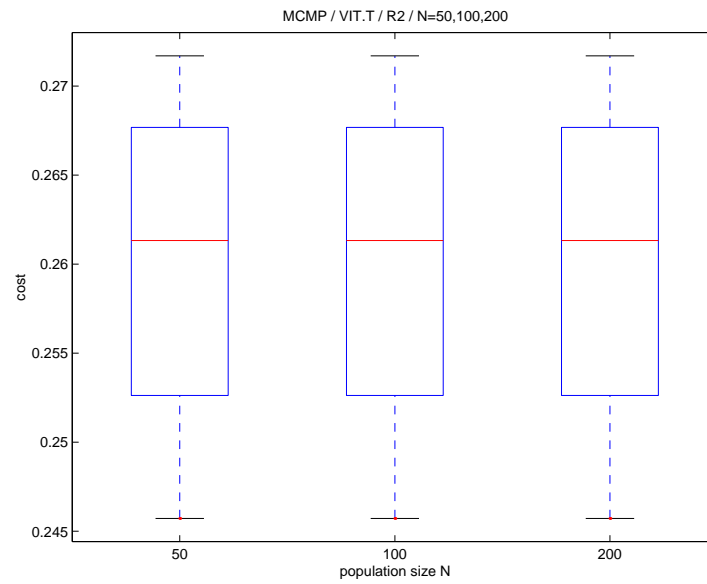


(b) FTS

Figure 9.20: Static heating with different population sizes—9.20(a) FIS and 9.20(b) FTS.



(a) VIT.I



(b) VIT.T

Figure 9.21: Dynamic heating with different population sizes—9.21(a) VIT.I and 9.21(b) VIT.T.

tions that can be explored for  $N = 50$  may be an explanation for the better performance. In contrast, the heating scheme FTS achieves better results when a larger population  $n = 200$  is used. For the dynamic heating schemes, the results seem to be less sensitive to the population size.

## 9.6 Discussion

Several trends in the experimental data are summarized below:

- The dynamic variants of the simulated heating technique outperformed the standard hybrid global/local search technique.
- When employing the standard hybrid method utilizing a fixed parameter  $p$ , an optimal value of  $p$  may be isolated and difficult to find in advance.
- Such optimal values of  $p$  depend on the application.
- When performing simulated heating, our experiments show that choosing the parameter range to lie around the best fixed  $p$  values yields better results than using the broadest range in most cases. However, using the broader range still produces good results, and this is the method most likely to be used in practice.
- The dynamic heating schemes show less sensitivity to this parameter range.
- Overall, the dynamic heating schemes performed better than the static heating schemes.
- The dynamic heating schemes were also less sensitive to the population size of the global search algorithm.

## 9.7 Conclusions

Efficient local search algorithms, which refine arbitrary points in a search space into better solutions, exist in many practical contexts. In many cases, these local search algorithms can be parameterized so as to trade off time or space complexity for optimization accuracy. We call these parameterized local search algorithms (PLSAs). We have shown that a hybrid PLSA/EA (parameterized local search/evolutionary algorithm) can be very effective for solving complex optimization problems. We have demonstrated the importance of carefully managing the run-time/accuracy trade-offs associated with EA/PLSA hybrid algorithms, and have introduced a novel framework of simulated heating for this purpose. We have developed both static and dynamic trade-off management strategies for our simulated heating framework, and have evaluated these techniques on the binary knapsack problem and two complex, practical optimization problems with very different structure. These problems have vast solution spaces, and underlying PLSAs that exhibit a wide range of accuracy/complexity trade-offs. We have shown that, in the context of a fixed optimization time budget, simulated heating better utilizes the time resources and outperforms the standard fixed parameter hybrid methods. In addition, we have shown that the simulated heating method is less sensitive to the parameter settings.

## Chapter 10

### Conclusions and Future Work

In this thesis we explored the implications of varying degrees of connectivity and contention in multiprocessor embedded systems for digital signal processing applications. A trade-off exists in such systems between cost/complexity and reduced resource contention (leading to higher performance). We presented techniques for analyzing these trade-offs, for making the most efficient use of available resources at a given design point, and for streamlining the system for a targeted set of applications.

The simplest and cheapest systems utilize a shared electrical bus. As explained in Chapter 4, the shared bus precludes an analytic expression for the system throughput, and simulation is required to get an accurate performance measurement. However, simulation is computationally expensive and it is undesirable to perform repeated simulations during an optimization. We developed a *period graph* model that can be used as a computationally efficient estimator for the throughput in these systems. We demonstrated the utility of this estimator by using it in a genetic algorithm and a simulated annealing algorithm for a voltage scaling application to reduce power.

With the additional expense of a hardware bus controller that imposes a global ordering of all communications, it is possible to remove the contention that results in the diffi-



cult analysis and to more fully optimize communication patterns in an application. This has been demonstrated to increase the performance and to be a useful cost/performance trade-off for several applications [101]. For highly parallel applications with more severe real-time constraints, however, the single bus becomes a bottleneck for communication between processors. In Chapter 5 we introduced a system architecture that utilizes optical fiber interconnects over multiple wavelengths. This enables multiple, simultaneous communications and increases the system throughput. In this architecture there is a controller for each communication wavelength, and we introduced a modification of the TPO heuristic [62] for determining optimal communication orderings for all the wavelengths. We quantified the performance improvement over the single bus controller for several applications.

A wide range of scheduling techniques for multiprocessor systems have been developed. However, these techniques typically assume a fixed communication network and do not systematically incorporate connectivity constraints. Connectivity constraints may be dictated by cost, area, or power constraints. Due to the power consumption characteristics of optical links, it is useful to restrict communication across them to low-hop transfers. Connectivity constraints cause existing multiprocessor scheduling methods to deadlock. In Chapter 6 we demonstrated a polynomial complexity algorithm for determining the set of feasible processors that will avoid schedule deadlock in a limited-hop schedule. We also introduced a useful metric, called communication flexibility, for the degree to which a given scheduling decision constrains future scheduling decisions (in the context of the given communication topology). We used this algorithm and the flexibility metric in conjunction with a standard dynamic list scheduling algorithm to effectively map several DSP applications across a wide range of interconnect topologies.

In Chapter 7 we explored the problem of deriving an interconnect network for a given application that minimizes the number of links required and maintains fanout constraints, while also satisfying the throughput or latency requirements of the application. This problem is important in today’s system-on-chip (SoC) designs as well as future SoC designs that might utilize optical interconnects. We described probabilistic and deterministic algorithms for interconnect synthesis. A key distinguishing feature of our technique is that we perform scheduling and interconnect synthesis together—existing interconnect synthesis algorithms assume a given application mapping exists before performing the interconnect synthesis. We demonstrated how the design space can be greatly reduced by considering graph isomorphism, and utilized an efficient graph isomorphism tests in our deterministic algorithm.

Most optimization problems that arise in hardware-software co-design are highly complex. The scheduling, interconnect synthesis, memory, and voltage scaling optimization problems investigated in this thesis all involve searching vast design spaces. In Chapter 8 we demonstrated that a hybrid PLSA/EA (parameterized local search/evolutionary algorithm) can be very effective for solving these complex optimization problems. We presented PLSAs for the voltage scaling, interconnect synthesis, and ordered transactions problems.

We demonstrated the importance of carefully managing the run-time/accuracy trade-offs associated with EA/PLSA hybrid algorithms, and introduced a novel framework of simulated heating for this purpose. We developed both static and dynamic trade-off management strategies for our simulated heating framework, and in Chapter 9 evaluated these techniques on the voltage scaling problem, a memory cost minimization problem, and the binary knapsack problem. Simulated heating experiments with the interconnect synthesis problem and the ordered transactions problem are two directions for future

work.

The PLSAs underlying these problems exhibit a wide range of accuracy/complexity trade-offs. We have shown that, in the context of a fixed optimization time budget, simulated heating better utilizes the time resources and outperforms the standard fixed parameter hybrid methods. In addition, we have shown that the simulated heating method is less sensitive to the parameter settings.

## **.1 Random Graph Generation Algorithm**

Sih's random graph generator [96] produces graphs with characteristics similar to those of many DSP benchmarks. We made several modifications to this algorithm to generate the random graphs used in this thesis.

First, before we add a random edge we first check (using Warshall's algorithm for transitive closure) that the edge will not introduce a cycle in the graph. Second, we input the number of nodes in the graph instead of the graph length. Third, we make the maximum fanout from each node an explicit input. This controls the amount of parallelism in the graph. Pseudo-code for the algorithm is given in Figures 1, 2, and 3.

**Algorithm A.1:** GENRANDOMGRAPH(

startNodes, numNodes, fanout, lowExecTime, highExecTime, lowIPCcost, highIPCcost

)

**procedure** CREATENODE(low, high)**comment:** Create and return a new node with random execution time  $\in [low, high]$ Create new node  $n$  $n.execTime \leftarrow \text{UNIFORMRANDOM}(low, high)$ **return** ( $n$ )**procedure** CONNECTNODES(src, snk, adjM)**comment:** Connect src node to snk node in adjacency matrix $adjM[src][snk] \leftarrow 1$ **procedure** EXTENDNODE(oldEndNode, endNodes, fanout)**comment:** Connect a newly created node to one of the endNodes $r \leftarrow \text{UNIFORMRANDOM}(1, fanout)$ **for** ( $i \leftarrow 1 \dots r$ )

<b>do</b>	{	$m \leftarrow \text{CREATENODE}(lowExecTime, highExecTime)$
		$\text{CONNECTNODES}(oldEndNode, m)$
		$endNodes.delete(oldEndNode)$
		$endNodes.add(m)$
}		

**procedure** CONVERGE(nodesToConverge, endNodes)**comment:** Cause some endNodes to all converge to a single node $p \leftarrow \text{CREATENODE}(lowExecTime, highExecTime)$ **for**  $i \leftarrow 1 \dots nodesToConverge.size()$ 

<b>do</b>	{	$nodesToConverge.deleteHead(h)$
		$endNodes.delete(h)$
		$\text{CONNECTNODES}(h, p)$
}		

 $endNodes.add(p)$ **procedure** DIVERGE(endNodes, num, V, divergedNodes)**comment:** randomly chosen endnode diverges out**for**  $i \leftarrow 1 \dots num$ 

<b>do</b>	{	$n \leftarrow \text{CREATENODE}(lowExecTime, highExecTime)$
		$\text{CONNECTNODES}(V, n)$
		$divergedNodes.add(n)$
}		

 $endNodes.delete(V)$ 

Figure 1: Pseudo-code for procedures used in the random graph algorithm.

**Algorithm A.1:** GENRANDOMGRAPH(

startNodes, numNodes, fanout, lowExecTime, highExecTime, lowIPCcost, highIPCcost

)

**procedure** DIVERGECONVERGE(endNodes, V, W, L, numAdded)

**comment:** Attach a structure which diverges and then converges to an endnode

$w \leftarrow \text{UNIFORMRANDOM}(2, W)$

nodeList  $\leftarrow \emptyset$

DIVERGE(endNodes, w, V, divergedNodes)

len  $\leftarrow$  choose randomly from  $[0 \dots L]$

**for**  $i \leftarrow 1 \dots w$

**do**  $\begin{cases} \text{divergedNodes.deleteHead}(h) \\ n \leftarrow \text{EXTENDNODE}(h, \text{endNodes}, \text{len}) \\ \text{nodeList.add}(n) \end{cases}$

CONVERGE(nodeList, endNodes)

numAdded  $\leftarrow w(\text{len} + 1) + 1$

**procedure** PICKRANDOMLY(nodeList, n)

**comment:** Create a random list of n nodes from nodeList

S  $\leftarrow \emptyset$

**while** (nodeList.size() < n)

$\begin{cases} p \leftarrow \text{nodeList.firstPtr} \\ r \leftarrow \text{UNIFORMRANDOM}(0, \text{nodeList.size}) \\ \text{for } i \in [1 \dots r] \\ \text{do } \begin{cases} p \leftarrow p.\text{next} \\ \text{if } p \notin S \\ \text{then } \begin{cases} S \leftarrow S \cup \{p\} \\ \text{nodeList.insert}(p) \end{cases} \end{cases} \end{cases}$

**procedure** RANDOMCONNECTION(adjM)

**comment:** Add a random edge that doesn't create a cycle

ok  $\leftarrow$  FALSE

**while** (ok = FALSE)

$\begin{cases} h \leftarrow \text{PICKRANDOMLY}(\text{allNodes}, 1) \\ t \leftarrow \text{PICKRANDOMLY}(\text{allNodes}, 1) \\ \text{do } \begin{cases} \text{TRANSITIVECLOSURE}(\text{adjM}) \\ \text{if } (\text{PATH}(h, t) = 0) \\ \text{then } \begin{cases} \text{ok} \leftarrow \text{TRUE} \end{cases} \end{cases} \end{cases}$

CONNECTNODES(h, t, adjM)

Figure 2: Pseudo-code for the random graph algorithm (continued from Figure 1).

**Algorithm A.1:** GENRANDOMGRAPH(

startNodes, numNodes, fanout, lowExecTime, highExecTime, lowIPCcost, highIPCcost

)

**main****for**  $i \leftarrow 1 \dots \text{startNodes}$ **do**  $\left\{ \begin{array}{l} u \leftarrow \text{CREATENODE}(\text{lowExecTime}, \text{highExecTime}) \\ \text{endNodes.add}(u) \end{array} \right.$  $n \leftarrow \text{startNodes}$ **while**  $(n < \text{numNodes})$ 

$$\left\{ \begin{array}{l} \text{actionNumber} \leftarrow \text{UNIFORMRANDOM}(0, 100) \\ \text{if } (\text{actionNumber} < 20) \\ \quad \left\{ \begin{array}{l} v \leftarrow \text{PICKRANDOMLY}(\text{endNodes}, 1) \\ \text{do } \left\{ \begin{array}{l} \text{EXTENDNODE}(v, \text{endNodes}, 1) \\ n \leftarrow n + 1 \end{array} \right. \\ \text{else if } (\text{actionNumber} < 40) \\ \quad \left\{ \begin{array}{l} c \leftarrow \text{UNIFORMRANDOM}(1, \text{fanout}) \\ \text{do } \left\{ \begin{array}{l} \text{convNodes} \leftarrow \text{PICKRANDOMLY}(\text{endNodes}, c) \\ \text{CONVERGE}(\text{convNodes}, \text{endNodes}) \\ n \leftarrow n + 1 \end{array} \right. \\ \text{else if } (\text{actionNumber} < 80) \\ \quad \left\{ \begin{array}{l} d \leftarrow \text{UNIFORMRANDOM}(1, \text{fanout}) \\ \text{do } \left\{ \begin{array}{l} u \leftarrow \text{PICKRANDOMLY}(\text{endNodes}, 1) \\ \text{DIVERGE}(\text{endNodes}, d, u, \text{divergedNodes}) \\ n \leftarrow n + d \end{array} \right. \\ \text{else if } (\text{actionNumber} < 100) \\ \quad \left\{ \begin{array}{l} u \leftarrow \text{PICKRANDOMLY}(\text{endNodes}, 1) \\ \text{do } \left\{ \begin{array}{l} e \leftarrow \text{UNIFORMRANDOM}(1, \text{fanout}) \\ \text{DIVERGECONVERGE}(\text{endNodes}, u, d, e, \text{numAdded}) \\ n \leftarrow n + \text{numAdded} \end{array} \right. \end{array} \right. \end{array} \right.$$
**for**  $i \leftarrow 1 \dots \text{numRandomConnections}$ **do**  $\{ \text{RANDOMCONNECTION}(\text{adjM})$ 

Figure 3: Pseudo-code for the random graph algorithm (continued from Figure 2).

## BIBLIOGRAPHY

- [1] A. R. Agarwal, R. Simoni, J. L. Hennessy, and M. A. Horowitz, *An evaluation of directory schemes for cache coherence*, Proceedings of the 15th International Symposium on Computer Architecture, June 1988, pp. 280–289.
- [2] Arvind and R. S. Nikhil, *Executing a program on the mit tagged-token dataflow architecture*, IEEE Transactions on Computers **39** (1990), no. 3, 300–318.
- [3] J. Backus, *Can programming be liberated from the von Neumann style? a functional style and its algebra of programs*, Communications of the ACM **21** (1978), no. 8, 613–641.
- [4] E. Balas and E. Zemel, *An algorithm for large zero-one knapsack problems*, Operations Research **28** (1980), 1130–1154.
- [5] N. Bambha and S. S. Bhattacharyya, *System synthesis for optically-connected, multiprocessors on-chip*, System on Chip for Real-time Systems (W. Badawy and G. A. Julien, eds.), Kluwer Academic Publishers, 2002, pp. 339–448.
- [6] ———, *Techniques for co-design of optically-connected embedded multiprocessors*, Proceedings of the Annual Workshop on High Performance Embedded Computing (Lexington, Massachusetts), September 2002, Extended abstract, pp. 97–99.



- [7] N. Bambha, S. S. Bhattacharyya, J. Teich, and E. Zitzler, *Hybrid search strategies for dynamic voltage scaling in embedded multiprocessors*, Proceedings of the International Workshop on Hardware/Software Co-Design (Copenhagen, Denmark), April 2001, pp. 243–248.
- [8] N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharyya, *Intermediate representations for design automation of multiprocessor DSP systems*, Journal of Design Automation for Embedded Systems **7** (2002), no. 4, 307–323.
- [9] N. K. Bambha and S. S. Bhattacharyya, *A joint power/performance optimization technique for multiprocessor systems using a period graph construct*, Proceedings of the International Symposium on System Synthesis (Madrid, Spain), September 2000, pp. 91–97.
- [10] ———, *A period graph throughput estimator for multiprocessor systems*, Tech. Report UMIACS-TR-2000-49, Institute for Advanced Computer Studies, University of Maryland at College Park, July 2000, Also Computer Science Technical Report CS-TR-4159.
- [11] N. K. Bambha, S. S. Bhattacharyya, and G. Euliss, *Design considerations for optically connected systems on chip*, Proceedings of the International Workshop on System on Chip for Real Time Processing (Calgary, Canada), June 2003, pp. 299–303.
- [12] N. K. Bambha, S. S. Bhattacharyya, J. Teich, and E. Zitzler, *Systematic integration of parameterized local search into evolutionary algorithms*, IEEE Transactions on Evolutionary Algorithms **8** (2004), no. 2, 137–155.

- [13] P. M. C. C. Barahona and J. R. Gurd, *Simulated performance of the Manchester multi-ring dataflow machine*, Proceedings of 2nd ICPC, September 1985, pp. 419–424.
- [14] B. Barrera and E. A. Lee, *Multirate signal processing in Comdisco's SPW*, Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, April 1991.
- [15] S. S. Bhattacharyya, *Compiling dataflow programs for digital signal processing*, Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, July 12 1994.
- [16] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Optimal parenthesization of lexical orderings for DSP block diagrams*, Proceedings of the International Workshop on VLSI Signal Processing, IEEE press (Sakai, Osaka, Japan), October 1995, pp. 177–186.
- [17] ———, *Software synthesis from dataflow graphs*, Kluwer, Norwell MA, 1996.
- [18] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Optimizing synchronization in multiprocessor DSP systems*, IEEE Transactions on Signal Processing **45** (1997), no. 6, 1605–1618.
- [19] T. Blickle, J. Teich, and L. Thiele, *System-level synthesis using evolutionary algorithms*, Tech. Report Technical Report 16, Swiss Federal Institute of Technology (ETH), Zurich, April 1996.
- [20] J. T. Buck, *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*, Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, September 1993.

- [21] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, *Ptolemy: A framework for simulating and prototyping heterogeneous systems*, International Journal of Computer Simulation **4** (1994), 155–182.
- [22] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, *Introduction to upc and language specification*, Tech. Report CCS-TR-99-157, Center for Computing Sciences, May 1999.
- [23] E. Carter, <http://www.taygeta.com/annealing/simanneal.html>, Tech. report, Taygeta Scientific, Inc., 2001.
- [24] A. P. Chandrakasan, S. Sheng, and R. W. Broderson, *Low-power CMOS digital design*, IEEE Journal of Solid-State Circuits **27** (1992), no. 4, 473–484.
- [25] L. Chao and E. Sha, *Scheduling data-flow graphs via retiming and unfolding*, IEEE Transactions on Parallel and Distributed Systems **8** (1997), no. 12, 1259–1267.
- [26] R. T. Chen, *Si CMOS process-compatible guided-wave multi-Gbit/sec optical clock distribution system for the Cray T-90 supercomputer*, Proceedings of Massively Parallel Processing with Optical Interconnections '97, IEEE CS Press, Los Alamitos, CA, 1997, pp. 10–24.
- [27] F. Commoner, *Deadlocks in petri nets*, Tech. Report CA-7206-2311, Massachusetts Computer Associates, Wakefield, MA, June 1972.
- [28] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, MIT Press, 1992.

- [29] A. Dasdan and R. K. Gupta, *Faster maximum and minimum mean cycle algorithms for system-performance analysis*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **17** (1998), no. 10, 889–899.
- [30] L. Davis, *Handbook of genetic algorithms*, Van Nostrand Reinhold, New York, 1991.
- [31] G. de Micheli, *Synthesis and optimization of digital circuits*, McGraw Hill, 1994.
- [32] J. B. Dennis, *First version data flow procedure language*, Tech. Report MAC Technical Memorandum 61, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1975.
- [33] ———, *Data flow supercomputers*, IEEE Computer **13** (1980), 48–56.
- [34] J. B. Dennis and D. P. Misunas, *A preliminary architecture for a basic data-flow processor*, Proceedings of 2nd ISCA, January 1975, pp. 126–132.
- [35] P. Eles, A. Doboli, P. Pop, and Z. Peng, *Scheduling with bus access optimization for distributed embedded systems*, IEEE Transactions on VLSI Systems **8** (2000), no. 5, 472–491.
- [36] J. Fan, B. Catanzaro, V. H. Ozguz, C. K. Cheng, and S. H. Lee, *Design considerations and algorithms for partitioning optoelectronic multichip modules*, Applied Optics **34** (1995), 3116–3127.
- [37] M. R. Feldman and C. C. Guest, *Interconnect density capabilities of computer generated holograms for optical interconnection of very large scale integrated circuits*, Applied Optics **28** (1989), 3134–3137.

- [38] T. Feo and M. Resende, *A probabilistic heuristic for a computationally difficult set covering problem*, Operations Research Letters **8** (1989), 67–71.
- [39] T. Feo, K. Venkatraman, and J. Burd, *A GRASP for a difficult single machine scheduling problem*, Computers and Operations Research **18** (1991), 635–643.
- [40] C. Fleurent and J. Ferland, *Genetic hybrids for the quadratic assignment problem*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science **16** (1994).
- [41] M. J. Flynn, *Very high-speed computing systems*, Proceedings of the IEEE **54** (1966), no. 12, 1901–1909.
- [42] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of np-completeness*, W. H. Freeman, 1991.
- [43] D. Goldberg, *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley, 1989.
- [44] D. E. Goldberg and S. Voessner, *Optimizing global-local search hybrids*, Proceedings of GECCO '99, vol. 1, 1999, pp. 220–228.
- [45] H. H. Goldstine, *The computer: From Pascal to von Neumann*, Princeton University Press, Princeton, NJ, 1972.
- [46] P. S. Guilfoyle, *Digital optical computing architectures for compute intensive applications*, Proceedings of International Conference on Optical Computing, 1994.
- [47] J. R. Gurd, C. C. Kirkham, and I. Watson, *The manchester prototype dataflow computer*, Communications of the ACM **28** (1985), no. 1, 34–52.

- [48] M.W. Haney, M.P. Christensen, and P. Milojkovic, *Description and evaluation of the fast-net smart pixel-based optical interconnection prototype*, Proceedings of the IEEE **88** (2000), no. 6, 819–828.
- [49] H. He, J. Xu, and X. Yao, *Solving equations by hybrid evolutionary computation techniques*, IEEE Transactions on Evolutionary Computation **4** (2000), no. 3, 295–304.
- [50] J. Henkel and R. Ernst, *A path-based technique for estimating hardware runtime in hw/sw-cosynthesis*, Proceedings of the Eighth International Symposium on System Synthesis, September 1995, pp. 116–121.
- [51] K. Hiraki, S. Sekiguchi, and T. Shimada, *Status report of SIGMA-1: A dataflow supercomputer*, Prentice Hall, 1991.
- [52] C-H Hwang and A.C. H. Wu, *A predictive system shutdown method for energy saving of event-driven computation*, Digest of Technical Papers, 1997 IEEE International Conference on Computer-Aided Design ICCAD-97, November 1997, pp. 28–32.
- [53] H. Ishibuchi and T. Murata, *Multi-objective genetic local search algorithm*, Proceedings of IEEE Conference of Evolutionary Computation (ICEC '96), 1996, pp. 119–124.
- [54] M. Ishikawa, *Optoelectronic parallel computing system with reconfigurable optical interconnection*, Critical Reviews **CR62** (1996), 156–175.
- [55] M. Kagawa, H. Tsukada, and M. Yoneda, *Optical add-drop multiplexers for metro/access networks*, (2003), no. 23, 59–64.

- [56] A. Kahn, C. McCreary, J. Thompson, and M. McArdle, *A comparison of multiprocessor scheduling heuristics*, Proceedings of 1994 International Conference on Parallel Processing vol. II, 1994, pp. 243–250.
- [57] M. Kalos and P. Whitlock, *Monte carlo methods*, Wiley-Interscience, New York, 1986.
- [58] R. M. Karp, *A characterization of the minimum cycle mean in a digraph*, Discrete Mathematics **23** (1978), 309–311.
- [59] R. M. Karp and R. E. Miller, *Properties of a model for parallel computation: Determinacy, termination, and queueing*, SIAM Journal of Applied Math **14** (1966), no. 6, 1390–1411.
- [60] S. Karzalis, S. Papadakis, and J. Theocharis, *Microgenetic algorithms as generalized hill-climbing operators for GA optimization*, IEEE Transactions on Evolutionary Computation **5** (2001), no. 3, 204–217.
- [61] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal **49** (1970), 291–307.
- [62] M. Khandelia and S. S. Bhattacharyya, *Contention-conscious transaction ordering in embedded multiprocessors*, Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors, July 2000, pp. 276–285.
- [63] R.K. Kostuk, J.W. Goodman, and L. Hesselink, *Optical imaging applied to microelectronic chip-to-chip interconnections*, Applied Optics (1985), 2851–2858.
- [64] D. Kreher and D. Stinson, *Combinatorial algorithms: Generation, enumeration, and search*, CRC Press, 1999.

- [65] A. V. Krishnamoorthy, P. J. Marchand, F. E. Kiamilev, and S. C. Esener, *Grain-size considerations for optoelectronic multistage interconnection networks*, Applied Optics **31** (1992), 5480–5507.
- [66] E. L. Lawler, *Combinatorial optimization*, Holt, Rinehart, and Winston, 1976.
- [67] E. A. Lee and J. C. Bier, *Architectures for statically scheduled dataflow*, Journal of Parallel and Distributed Computing **10** (1990), 333–348.
- [68] E. A. Lee and S. Ha, *Scheduling strategies for multiprocessor real-time DSP*, Proceedings of Globecom, November 1989.
- [69] E.A. Lee and D.G. Messerschmitt, *Static scheduling of synchronous dataflow programs for digital signal processing*, IEEE Transactions on Computers **36** (1987), no. 1, 24–35.
- [70] T.F. Leighton, *Introduction to parallel algorithms and architectures; arrays, trees, and hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.
- [71] Y. Li, E. Towe, and M. W. Haney, *Special issue on optical interconnections for digital systems*, Proceedings of the IEEE **88** (2000), no. 6, 723–727.
- [72] Y. T. S. Li and S. Malik, *Performance analysis of embedded software using implicit path enumeration*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **16** (1997), no. 12, 1477–1487.
- [73] D. Marculescu, *On the use of microarchitecture-driven dynamic voltage scaling*, Proceedings of ISCA 2000, 2000.
- [74] P. Marwedel and G. Goosens, *Code generation for embedded processors*, Kluwer Academic Publishers, 1995.



- [75] M. Masanovic, E. Skogen, J. Barton, V. Lal, K. Blumenthal, and L. Coldren, *Demonstration of monolithically-integrated InP widely-tunable laser and SOA-MZI wavelength converter*, Proceedings of International Conference on Indium Phosphide and related materials, May 2003, pp. 289–291.
- [76] N. McArdle and M. Ishikawa, *Experimental realization of a smart-pixel optoelectronic computing system*, Proceedings Massively Parallel Processing with Optical Interconnects '97 (1997), 190–195.
- [77] N. McArdle, M. Naruse, and M. Ishikawa, *Optoelectronic parallel computing using optically interconnected pipelined processing arrays*, IEEE Journal on Selected Topics in Quantum Electronics **5** (1999), no. 2, 250–260.
- [78] B. McKay, *Nauty user's guide*, Tech. Report Technical Report TR-CS-90-02, Australian National University, 1990.
- [79] J. D. Meindl, *Low power microelectronics: Retrospect and prospect*, Proceedings of the IEEE **83** (1995), no. 4, 619–635.
- [80] P. Merz and B. Freisleben, *A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem*, Proceedings of International Conference on Evolutionary Computation (CEC '99), 1999, pp. 2063–2070.
- [81] T. Miyazaki, *The complexity of McKay's canonical labeling algorithm*, Groups and Computation II **28** (1997), 239–256.
- [82] P. K. Murthy, E. G. Cohen, and S. Rowland, *System Canvas: A new design environment for embedded DSP and telecommunications systems*, Proceedings of the Ninth International Symposium on Hardware/Software Codesign, April 2001, pp. 54–59.

- [83] W. Nakayama, *Heat-transfer engineering in systems integration—outlook for closer coupling of thermal and electrical designs of computers*, IEEE Transactions on Components, Packaging, and Manufacturing Technology **Part A:18** (1995), 818–826.
- [84] H. M. Ozaktas and J. W. Goodman, *Elements of a hybrid interconnection theory*, Applied Optics **33** (1994), 2968–2987.
- [85] D. A. Patterson and J. L. Hennessy, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann, San Francisco CA, 1990.
- [86] J. L. Peterson, *Petri net theory and modeling of systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [87] D. Pisinger, *Core problems in knapsack algorithms*, Tech. Report Technical Report 94/26, DIKU, University of Copenhagen, Denmark, June 1994.
- [88] ———, *An expanding-core algorithm for the exact 0-1 knapsack problem*, European Journal of Operations Research **87** (1995), 175–177.
- [89] F. Ratovelomanana, N. Vodjdani, A. Enard, G. Glastre, D. Rondi, R. Blondeau, C. Joergensen, T. Durhuus, B. Mikkelsen, K. Stubkjaer, A. Jourdan, and G. Soulage, *An all-optical wavelength-converter with semiconductor optical amplifiers monolithically integrated in an asymmetric passive Mach-Zehnder interferometer*, IEEE Photonics Letters **7** (1995), no. 9, 992–994.
- [90] R. Reiter, *Scheduling parallel computations*, Journal of the Association for Computing Machinery **15** (1968), no. 4, 590–599.
- [91] S. Reiter and G. Sherman, *Discrete optimizing*, Journal of the Society for Industrial and Applied Mathematics **13** (1965), 864–889.

- [92] M. Ryan, J. Debusse, G. Smith, and I. Whittle, *A hybrid genetic algorithm for the fixed channel assignment problem*, Proceedings of GECCO '99, vol. 2, 1999, pp. 1707–1714.
- [93] V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessors*, The MIT Press, Cambridge, MA, 1989.
- [94] C. Seo and A. Chatterjee, *A cad tool for system-on-chip placement and routing with free-space optical interconnect*, Proceedings of IEEE International Conference on Computer Design (ICCD'02) (2002), 24–29.
- [95] H. Sethu, C. B. Strunkel, and R. F. Stucke, *IBM RS/6000 SP interconnection network topologies for large systems*, Proceedings of the International Conference on Parallel Processing, Minneapolis, MN, August 1998.
- [96] G. C. Sih, *Multiprocessor scheduling to account for interprocessor communication*, Ph.D. thesis, Department of EECS, U. C. Berkeley, 1991.
- [97] G. C. Sih and E. A. Lee, *A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures*, IEEE Transactions on Parallel and Distributed Systems **4** (1993), no. 2, 75–87.
- [98] J. Silc, B. Robic, and T. Ungerer, *Asynchrony in parallel computing: From dataflow to multithreading*, Tech. Report CSD-97-4, Institute Jozef Stefan, Computer Systems Department, Ljubljana, Slovenia, September 1997.
- [99] S. Skiena, *Graph isomorphism, implementing discrete mathematics: Combinatorics and graph theory with mathematica*, Addison-Wesley, Reading MA, 1990.
- [100] P. W. Smith, *On the physical limits of digital optical switching and logic elements*, Bell System Technical Journal **61** (1982), no. 8.

- [101] S. Sriram and S. S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*, Marcel Dekker Inc., 2000.
- [102] S. Sriram and E. A. Lee, *Determining the order of processor transactions in statically scheduled multiprocessors*, Journal of VLSI Signal Processing **15** (1997), no. 3, 207–220.
- [103] M. Srivastava, A. P. Chandrakasan, and R. W. Broderon, *Predictive system shutdown and other architectural techniques for energy efficient programmable computation*, IEEE Transactions on VLSI Systems **4** (1996), no. 1, 42–55.
- [104] N. Takahashi and M. Amamiya, *A data flow processor array system: Design and analysis*, Proceedings of 10th ISCA, June 1983, pp. 243–250.
- [105] J. Teich, T. Blickle, and L. Thiele, *An evolutionary approach to system-level synthesis*, Proceedings of the 5th International Workshop on Hardware/Software Codesign, March 1997, pp. 167–171.
- [106] M. Vazquez and D. Whitley, *A hybrid genetic algorithm for the quadratic assignment problem*, Proceedings of GECCO 2000, 2000.
- [107] I. Watson and J. R. Gurd, *A prototype data flow computer with token labelling*, Proceedings of the National Computer Conference, June 1979, pp. 623–628.
- [108] M. Wu and D. Gajski, *Hypertool: A programming aid for message-passing architectures*, IEEE Transactions on Parallel and Distributed Systems **1** (1990), no. 3, 101–119.
- [109] T. Yang and A. Gerasoulis, *A fast scheduling algorithm for DAGs on an unbounded number of processors*, Proceedings of the 5th ACM International Conference on Supercomputing, 1991, pp. 633–642.

- [110] <http://www.diku.dk/pisinger/codes.html>.
- [111] E. Zitzler, J. Teich, and S. S. Bhattacharyya, *Evolutionary algorithm based exploration of software schedules for digital signal processors*, Proceedings of GECCO '99, vol. 2, 1999, pp. 1762–1769.
- [112] ———, *Multidimensional exploration of software implementations for DSP algorithms*, Journal of VLSI Signal Processing **24** (2000), no. 1, 83–98.
- [113] ———, *Optimizing the efficiency of parameterized local search within global search: A preliminary study*, Proceedings of the Congress on Evolutionary Computation, July 2000, pp. 365–372.