Distributed Signal Processing

by

Li Lee

B.S., Massachusetts Institute of Technology (1996)
M.Eng., Massachusetts Institute of Technology (1996)

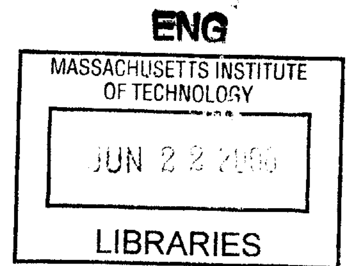
Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2000


ENG
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 2 2 2000
LIBRARIES

© Massachusetts Institute of Technology 2000. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
March 10, 2000

Certified by ....
Alan V. Oppenheim
Ford Professor of Electrical Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Distributed Signal Processing

by

Li Lee

Submitted to the Department of Electrical Engineering and Computer Science
on March 10, 2000, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

This thesis studies system-level issues arising from implementing Digital Signal Processing (DSP) applications on highly heterogeneous and dynamic networks of processors. Compared with traditional DSP computation platforms such as microprocessor chips, distributed networks of processors offer many potential advantages such as resource sharing and fault tolerance. To realize these benefits, however, new issues in system and algorithm design must be resolved. The heterogeneous and time-varying composition of such environments reduces the efficacy of traditional methods for algorithm design and selection.

This thesis develops a framework, called algorithm adaptation, which enables the system to make the ultimate selection of the signal processing algorithms applied to a data stream requiring processing. Our formulation of this strategy derives from a system model which accomodates a wide variety of processor types, from Application-Specific Integrated Circuits (ASICs) to general purpose microprocessors. The system is presented with the processing options using a DSP task description language which simultaneously expresses concurrencies and equivalences within DSP algorithms. A graphical visualization of the task descriptions leads to an interesting interpretation of algorithms as similar to communication networks, as well as mathematical formulations for dynamically choosing an execution path according to packet-wise or system-wide optimization criteria. To better quantify the effects of allowing execution paths to be dynamically chosen by the system, the thesis formulates a measure of the system capacity. A software simulation environment demonstrates that the algorithm adaptation framework increases system capacity and reduces processing delay in dynamic networks of heterogeneous processors.

Thesis Supervisor: Alan V. Oppenheim
Title: Ford Professor of Electrical Engineering

Acknowledgments

I would like to thank my thesis advisor, Prof. Alan Oppenheim, for his advice, encouragement, and patience throughout the journey that has culminated in this thesis. Al exemplifies dedication and leadership in everything he does, and I am grateful to have had the opportunity to study under his tutelage. I will never forget that "crisis" consists of both "danger" and "opportunity".

I would also like to thank my thesis committee, Prof. Dimitri Bertsekas and Prof. George Verghese for their time and valuable suggestions during the thesis writing process.

I have been extremely fortunate to have had the friendship and mentoring of Richard Rose of AT&T and Alan Berenhaum of Lucent over the years. Rick spent many hours teaching me how to write my first papers and give my first presentations–both "lifelines" in the career of a researcher. Alan opened my eyes about many aspects of life, from research to travel to cuisine to books, all of which I hope to fully explore in the future.

My friends kept my spirits high and gave me endless happy times. Thanks to Clara Poon, Desmond Lim, Huan Yao, Alice Wang, Jiang-Ti Kong, Brian Chen, Kush Gulati, Vivek Nadkarni, Magda Leuca, Soosan Beheshti, & Maya Said, for dimsum runs, cards, cooking sessions, zephyrs.

To "00": You are my best friend, my love, and my partner in all things. With you by my side, I feel like I can fly. I am so happy that we found each other.

To my parents and sister: Thank you for your boundless love, understanding, and humor. This thesis is dedicated to you as a small token of my gratitude.

Contents

1 Introduction **11**

1.1 Problem Description 11

1.2 Approach 12

1.2.1 Functional Equivalence 13

1.2.2 Expressivity 13

1.3 Contributions 14

1.4 Thesis Outline 15

2 Background **17**

2.1 Distributed Computing Systems 17

2.2 Resource Allocation 20

2.2.1 Distributed Computing Systems 20

2.2.2 Flexible Manufacturing Systems 22

2.2.3 Communications Networks 23

2.3 Summary 25

3 System Model **27**

3.1 System Description 27

3.1.1 Heterogeneity 28

3.1.2 Time Variance 28

3.1.3 DSP Focus 29

3.2 System Model 30

3.2.1 Collection of Primitive Operators 30

3.2.2 Object-Oriented Data Blocks 31

3.3 Examples of Computing Environments 32

3.3.1 Internet 32

3.3.2 DSP Network Chips 33

3.3.3 FPGA Networks 34

3.4 Summary 34

4 Algorithm Specification **35**

4.1 Functional Equivalence of Algorithms 35

4.2 Algorithm Description Language 38

4.2.1 Notation 38

4.2.2 Concurrency Operations 40

4.2.3 Discussion 42

4.3 Algorithm Graph 42

4.3.1 Construction 43

4.3.2 Interdependent Decision Points 44

4.3.3 Examples 45

4.3.4 Terminology, Assumptions, and Notation 47

4.3.5 Decomposition 48

4.3.6 Considerations of Validity 50

4.4 Algorithm Graphs Compared to Communication Networks 52

4.4.1 Physical Layer 53

4.4.2 Data Link Control Layer 53

4.4.3 Network Layer 54

4.4.4 Transport Layer 56

4.5 Summary 56

5 Algorithm Adaptation **59**

5.1 Motivations 59

5.2 Minimum Cost Packet-wise Adaptation 61

5.2.1 Notation 62

5.2.2 Rules 63

5.2.3 Example 64

5.2.4 Discussion 67

5.3 Optimal System-wide Adaptation 68

5.3.1 Formulation 68

5.3.2 Numerical Solution 70

5.3.3 Discussion 72

5.4 Summary 72

6 System Capacity **75**

6.1 Flow Decomposition 75

6.1.1 Notation 76
6.1.2 Path Flow to Arc Flow Conversion 77
6.1.3 Arc Flow to Path Flow 78
6.2 System Capacity 80
6.2.1 System Model 80
6.2.2 Single Algorithm Capacity 81
6.2.3 Multiple Algorithm Capacity 82
6.3 System Design with Capacity Considerations 86
6.3.1 Single Algorithm Case 86
6.3.2 Simultaneous Capacity Requirements 88
6.3.3 Uncertain Capacity Requirements 90
6.4 Summary 92

7 Simulation Environment **95**
7.1 Objects and Input/Output 95
7.1.1 Input 95
7.1.2 Network 96
7.1.3 Data Blocks 96
7.1.4 Output 97
7.2 Handling Concurrency 97
7.2.1 Example 98
7.3 Handling Equivalence 101
7.3.1 Cost Function 101
7.3.2 Implementation of Packet-wise Minimum Cost Adaptation 103
7.3.3 Implementation of System-wide Optimization 103
7.4 Simple Experiments 105
7.4.1 Simulated Network 105
7.4.2 Steady-State Allocations 105
7.5 Summary 108

8 Algorithms for Synthetic Aperture Radar **109**
8.1 Fundamentals of Radar Imaging 110
8.1.1 Microwave imaging 110
8.1.2 Terminology 111
8.1.3 Assumptions 112
8.2 Range Resolution and Pulse Compression 113
8.2.1 Matched Filtering 114

8.2.2 Deramp FFT 115
8.3 Stripmap Mode SAR 116
8.3.1 Imaging Geometry and Notation 116
8.3.2 Point Target Response 117
8.3.3 Range-Doppler Algorithm 119
8.3.4 ω-k Migration Algorithm 121
8.3.5 Chirp-Scaling Algorithm 122
8.4 Functional Equivalence Among SAR Algorithms 125
8.5 Summary 126

9 Simulation with SAR Algorithms 127
9.1 Experimental Setup 127
9.2 System Behavior 128
9.3 Reaction to Sudden Changes 130
9.3.1 Reaction to Processor Failure 130
9.3.2 Reaction to Functionality Change 131
9.3.3 Reaction to Change in Workload 135
9.4 Summary 135

10 Summary and Future Directions 137
10.1 Conclusions 137
10.2 Future Directions 139
10.2.1 System Architecture Design 139
10.2.2 System Modeling 140
10.2.3 System Operation 140

Chapter 1

Introduction

With the growing prevalence of computer networks, the ability to efficiently exploit the computing power of the entire network becomes increasingly attractive. Compared with traditional Digital Signal Processing (DSP) computation platforms such as microprocessor chips, distributed networks offer many potential advantages such as resource sharing and fault tolerance. At the same time, however, distributed networks raise new challenges in system and algorithm design due to their heterogeneous and time-varying composition. This thesis establishes a framework for DSP algorithm and computing environment design for distributed networks of processors. This chapter motivates the essential features of this framework by specifying the special characteristics of the class of distributed systems under study, and the desirable operating characteristics which we seek to establish in such environments. We then present an overview of our approach, and describe the contributions of the thesis.

1.1 Problem Description

A *distributed computing system* is one in which the programs and/or the data which the programs operate on are spread out geographically over many processors. Because there is generally no reliance on any single resource in such systems, their potential benefits include resource sharing, fault tolerance, and scaleability. The subject of distributed computing has been an active research area in the computer science community for over 30 years.

The existing works on distributed computing can be roughly categorized into two broad classes: distributed systems and distributed algorithms. The realization of the benefits of distributed computing requires that the system provide certain services, and the methods to provide these services transparently, without the help of the user, are the focus of works in the first category. For example, when resources are shared, the system must have a fair and efficient resource allocation scheme. Similarly, a fault-tolerant system must have provisions to recover or back-up data or processing in

the event of failures.

The second class of research is concerned with algorithms which are intended to be run on distributed systems. These algorithms are cooperative in the sense that their execution requires the work of many processors. A large subset of such algorithms consists of those which implement the services provided in distributed computing systems. For example, routing algorithms are crucial in the overall design of communications networks [6]. A second subclass consists of algorithms especially designed to be run on top of such environments, taking advantage of the multiprocessor nature of the environment [5]. Depending on the nature of the problem, the design of these algorithms involves many issues, including correctness, rate of convergence, and bandwidth and computation efficiency [5] [28].

The goal of this thesis is to explore developments to extend distributed heterogeneous and dynamic computing environments to facilitate signal processing applications. The computing environment is assumed to be a network of processors, and moreover, the devices and communications links connecting them are heterogeneous and dynamic. Heterogeneity refers to the condition that the processor and link may differ in capability, and dynamic refers to the condition that they may fail and recover at any time. While there are many interesting issues involved in the study of signal processing algorithms for such environments, this thesis addresses the problem from a system perspective: In a heterogeneous and dynamic distributed system, what system characteristics and services would be helpful to the signal processing community, and how can the system provide these services?

1.2 Approach

A heterogeneous and dynamic computing environment poses an interesting problem to the signal processing engineer. Traditionally, the design and selection of a particular signal processing algorithm greatly depend on predictions of the bottleneck of the computational platform. For example, while time and memory usage are the criteria of interest for many microprocessor platforms, the power consumption of an algorithm is the optimization criterion of choice in low-power systems. In the context of heterogeneous distributed systems, an algorithm which is highly efficient on a certain type of processor may be very undesirable on another type. In a harsher situation, the type of processor suitable for one algorithm may not even be available at the time of processing. Finally, the bottleneck of the system may not reflect that of any individual processor. For example, the bottleneck of a large system of processors may not be processing complexity, even when each of the processors are limited in processing capabilities.

Because of this consideration, this thesis proposes a new system service called *algorithm adaptation*, through which the system selects and adapts the algorithm used to process a signal dy-

namically. The proposed framework takes advantage of two characteristics of signal processing algorithms. First, there exists many classes of *functionally equivalent* signal processing algorithms which accomplish basically the same processing objectives, albeit with different tradeoffs; the existence of these equivalences naturally presents the system with a variety of options. Second, signal processing algorithms can often be expressed as a composition of basic blocks, allowing the development of an efficient method for specifying functionally equivalent algorithms. The following discussion elaborates on these two characteristics.

1.2.1 Functional Equivalence

In this thesis, two algorithms are functionally equivalent with respect to a given data set if the qualities of their outputs are equally acceptable to the user. This definition coincides with statements of processing objectives of the form: "Given the data set and its known characteristics, process the data to return the desired information with quality measure of at least x." Depending on the type of the algorithm, the quality measure may refer to factors such as probability of error, compression ratio, or signal-to-noise ratio. In particular, functionally equivalent algorithms do not necessarily produce the same outputs when given the same inputs.

Signal processing algorithms can be functionally equivalent to others through several means. On the most direct level, equivalence can be established simply by re-arrangements of the mathematical operations which compose the algorithm; such equivalent algorithms produce identical results for all data sequences. A simple example is the operation of linear time-invariant filtering, which can be performed by convolution in the time domain or multiplication in the frequency domain.

More generally, however, functional equivalences are due to algorithmic development work which has resulted in classes of algorithms offering different types of tradeoffs. For example, the literature describes a variety of lossless compression algorithms which achieve different compression ratios while incurring different processing costs. In this case, all algorithms achieving or exceeding the given objective compression ratio are functionally equivalent. The functional equivalence among algorithms may further depend on the characteristics of the input data. As a simple example, in channel equalization for a time-invariant communications channel, adaptive and non-adaptive algorithms are functionally equivalent to each other; such would not be true if the channel were time-varying.

1.2.2 Expressivity

The brief discussion above suggests that functional equivalences among signal processing algorithms exist frequently due to the highly mathematical nature of both their formulations and their specifications. It turns out that this characteristic also facilitates the design of a method to specify a set of functionally equivalent algorithms efficiently. Specifically, it makes possible the composition

of algorithms in terms of basic building blocks. On the lowest level, these building blocks can be the basic mathematical operations, along with operations for shifting sequences. On a higher level, the building blocks can consist of signal processing *primitives*, such as Fourier transforms, convolutions, and so on. The expression of algorithms as compositions of these higher level building blocks is an efficient way to specify them. Given such expressions of a set of functionally equivalent algorithms, the system can analyze and choose among them, and then generate or find the relevant code implementing each step of the selected algorithm.

To summarize, the chosen approach of algorithm adaptation reflects the two distinguishing characteristics of the problem addressed in the thesis. The first is the heterogeneous and dynamic nature of the system, which makes a dynamic choice of algorithms attractive. The second is our special focus on signal processing, in which the mathematical nature of the algorithms makes them well-suited for algorithm adaptation. In the resulting system, the processing instructions accompanying the input data consist of a set of functionally equivalent algorithms. These instructions are dynamically analyzed and the execution paths chosen according to factors such as processor loads.

1.3 Contributions

Many issues are involved in the proper design of a distributed signal processing system. The scope of the thesis does not include the many tasks which are common to other types of distributed computing systems, such as communications, failure recovery, security, and system maintenance. We assume that the resolutions of those issues are similar to those of other systems. Rather, the main contributions of the thesis relate to our proposal to investigate system design issues which would be especially beneficial to the signal processing community. In particular, our exploration of algorithm adaptation has lead to four contributions.

First, the thesis establishes a system model highlighting the processors, their functionalities, and their characteristics. The processing network is conceptualized as a collection of specialized virtual signal processors whose functionalities are the primitive building blocks of algorithms executed in the system. The physical manifestation of the virtual processors is unspecified. Hence a virtual processor may reside on one or multiple physical processors, and each physical processor may behave as several virtual processors. In the thesis we use this model to capture the essence of dynamic and heterogeneous processing environments, and to facilitate our study of the adaptation of DSP algorithms to the environment.

Using the system model, the second contribution of the thesis is a framework for a distributed signal processing system implementing algorithm adaptation. We present a language for expressing signal processing algorithms which efficiently highlights the similarities and differences among functionally equivalent alternatives. This language leads naturally to a graphical representation of

algorithms with interesting interpretations. Additionally, we formulate and characterize methods for dynamically and optimally choosing among the algorithms to minimize cost functions such as packet delay or system congestion.

Next, extending the interpretation of algorithms as graphs, we derive the theoretical upper bound of the maximum rate at which a system of processors can process given sets of algorithms. We show that when the input rates fall below this bound, there exists an execution path allocation which guarantees that the processing limits of every primitive operator is respected. On the other hand, when the input rate exceeds the system capacity, every execution path allocation exceeds the processing limits of some primitive operator. Such analysis would be helpful for the understanding and comparison of the performance of algorithm adaptation strategies.

The final contribution of the thesis is a software simulation of a distributed signal processing system. The simulation implements algorithm adaptation and allows practical evaluation of the system response to events such as processor failures and sudden changes in processing load. Experiments using image formation algorithms for synthetic aperture radar (SAR) data are presented.

1.4 Thesis Outline

Chapter 2 relates this work to the existing developments on distributed computing systems, particularly in the computer science community. Through our discussion, we show that this thesis studies computation environments and scenarios which have been excluded from current works. Additionally, an overview of the current approaches to resource allocation in scenarios such as distributed computing systems, flexible manufacturing systems, and data communications networks is presented.

Chapter 3 presents a system model of the computing environment. The first part of the chapter offers a detailed description of the computing environment, along with our assumptions about its operation. A system model which simultaneously captures its salient characteristics and allows tractable analysis is then presented. Finally, examples of such environments are discussed to further clarify the system description and model.

Chapter 4 describes conceptualizations and constructions related to signal processing algorithms. After the concept of functional equivalence is discussed in depth, a language for specifying sets of the functionally equivalent algorithms is presented. A graphical representation of the language leads to the key construct of an algorithm graph. The parallelism between the structures of algorithm graphs and communications networks is then explored to reveal some interesting interpretations.

Chapter 5 presents two strategies for algorithm adaptation. The first method, based on the adaptive shortest path paradigm in data packet routing, chooses the minimum cost execution path for each data block independently. The second method, based on the optimal routing technique,

uses a system-wide optimization criterion. The algorithm adaptation problem is formulated as a constrained nonlinear optimization problem in which measures of system congestion is minimized. The numerical solutions for the optimization problem can be obtained iteratively using a gradient projection algorithm.

Chapter 6 extends the use of algorithm graphs to the derive the concept of the system capacity with respect to single and multiple algorithms. The methodology can be applied to find the maximum rate at which a given system can accomplish a given processing objective, and the "bottleneck" of the system which restricts its capacity. Based on the definition of system capacity, we present formulations to find the minimum set of processors and their functional distributions required to meet given system capacity specifications. Such analysis would be helpful to the system designer evaluating and comparing systems in terms of their processing capacity.

Chapters 7 through 9 present the simulation portion of this work. First, Chapter 7 describes the structure and explains the capabilities of the simulation environment. Since the experimental work is related to image formation algorithms for synthetic aperture radar (SAR), Chapter 8 offers an overview of these algorithms. Finally, Chapter 9 presents an experimental study involving the processing of SAR data on a dynamic network of heterogenous processors of the type we envision in the thesis.

Chapter 10 concludes with summarizing remarks and suggestions for future work.

Chapter 2

Background

In any research area, it is important to understand the current literature before undertaking new explorations and proposing new approaches. This is especially true of this thesis, because it is at least nominally reminiscent of the topic of distributed computing systems, on which there already exists a large body of works produced by the computer science community. Therefore, Section 2.1 reviews and describes the common threads within these existing works. Through this discussion, we show that the problem and the solution presented in this thesis differ dramatically from those of the current sets of works from the computer science community.

While our perspective represents a paradigm shift from the existing literature, we use many mathematical and optimization tools found in contexts such as job scheduling in distributed computing systems and data packet routing in telecommunication networks. Section 2.2 reviews literature covering the formulations and solutions to resource allocation problems in distributed computing systems, flexible manufacturing systems, and communications systems.

2.1 Distributed Computing Systems

Distributed computing systems are built on networks and processors and designed to be capable of handling a wide range of computational demands. Because of their potential for becoming highly cost effective, efficient, fault tolerant, and flexible, they have attracted much attention from the computer science community for over 30 years [45]. Cost effectiveness stems from the possibility of resource sharing; because many users can have access to both software and hardware resources over the network, unnecessary replications can be eliminated. Execution efficiency is possible because multiple processors can cooperate to solve a single problem. Fault tolerance is achievable through a few means: the data and control structures are replicable; computations at different processors can be checked against each other; and a system which is geographically distributed is less susceptible to location-specific outages. Finally, and perhaps most importantly, the overall system can adapt

to short-term changes in demand, processor failures and additions, as well as longer term changes in overall system architecture.

While all of the aforementioned benefits are highly desirable, they cannot be realized without careful design. When resources can be shared over the network, it becomes crucial to address issues such as security, fair access, and resource allocation. The cooperative execution of processes among multiple processors involves communications network design, algorithm design, and even programming language design. Fault tolerance schemes require data, control, or processes to be replicated across the network, which may pose problems for efficiency. Finally, the mechanisms used to achieve the above benefits must respond appropriately to changes so that the overall system is able to adapt gracefully to changing operating conditions.

Research in distributed computing systems is directed at methodologies to enable distributed systems to realize their potential benefits. With so many issues at hand, it is virtually impossible to review the entire body of work, which includes such topics as distributed operating systems, distributed programming languages, theory of distributed computing, fault tolerant system design, distributed real-time systems, and distributed applications. Within each topic, the literature can be further classified according to the assumptions on the hardware configurations and available levels of service. For example, the hardware can be *homogeneous* [12] [24], consisting of processors of exactly the same type, or *heterogeneous* [47] [38]. The network configuration can be *static* [14] [26], not changing during normal system operation, or *dynamic* [43]. The system can be *single-user* [37], providing service to a single job demand at a time, or it can be *multi-user* [50].

These broad ranges of work do share some common perspectives and assumptions, however. Among them, the most significant is perhaps the perspective on what exactly constitutes a distributed computing system. The views of the field are particularly well articulated in [17], where the author writes that distributed computing systems must have the following five characteristics:

1. A *multiplicity* of general purpose resource components, including both physical and logical resources, that can be assigned to specific tasks on a dynamic basis. For example, processors must not be so specialized that they are permanently binded to certain system tasks.

2. A *physical distribution* of these physical and logical components of the system interacting through a communication network. Processors must be able to pass messages to each other, and any resource must be allowed to refuse the transfer or acceptance of a message based on its own knowledge of its own status.

3. A *high-level operating system* that unifies and integrates the control of the distributed components.

4. *System transparency*, permitting services to be requested by name only. The user should be able to develop routines and programs and handle data bases just as if he were communicating

with a single, centralized system.

5. *Cooperative autonomy*, characterizing the operation and interaction of both physical and logical resources.

While this thesis accepts the premises of most of these characterizations, the first characteristic reveals one essential difference between prior work and this thesis. In our work, we do not assume that only general purpose processors can be dynamically assigned to different processing tasks. Because signal processing algorithms can be expressed as a composition of primitive operations, a specialized processor can be used to accomplish a part of many different processing tasks. Conversely, because of the flexibility afforded by functional equivalences among multiple algorithms, a permanent binding between a processing task and a specific processor is not required. Hence, even if each processor is specialized, the framework proposed in this thesis allows the system to achieve the type of generality and "dynamic binding" which has been assumed to be possible only with general purpose processors.

Because of the relaxation of this assumption, previous work on dynamic and heterogeneous distributed computing systems does not apply in the scenarios we envision. A survey of papers from the Heterogeneous Computing Workshop shows that heterogeneity of processors simply means a lack of homogeneity; all of the processors are immediately assumed to be general purpose [19]. In such systems, any job can be assigned or migrated to any processor at any time, and hence one major area of investigation involves the "optimal" partitioning of processing tasks into subtasks and the subsequent matching and scheduling of these subtasks onto processors [7]. Other issues include providing for system-level services such as interoperability among processors, checkpointing, task migration, and networking [36].

While these works are clearly crucial to the successful operation of broad classes of existing networks, in this thesis we propose the investigation of networks which are heterogeneous to a greater degree. One important motivation for this direction is the growing availability of devices such as FPGAs, ASICs, and DSPs, which do not fall within the category of general purpose processors. At the same time, the ability to include such devices enriches the set of networks under consideration. Moreover, as this thesis shows, with the proper design, the inclusion of specialized processors does not limit the generality of the overall system.

Another assumption made in the current literature is that each submitted processing task has only one possible execution path; this is such a common and deeply-engrained preconception that it is practically invisible. However, in the signal processing area, often there are many algorithms which are capable of accomplishing a processing objective. Therefore, a second significant difference between this thesis and prior research on distributed computing systems is that we take advantage of this additional degree of freedom to compensate for the fact that some of the processors may

be highly specialized. This realization forms the backbone of the algorithm adaptation schemes described in Chapter 5.

Of course, a subset of issues involved in the highly heterogeneous and dynamic environments that we envision can be resolved through the solutions already proposed in the existing literature; examples include security, naming, and messaging scheme. However, because the differences between our work and previous works are at such fundamental levels, we believe that this thesis relates only superficially to studies on distributed computing systems in the computer science community. It is the intent of the thesis to focus on those issues which newly arise due to the new assumptions that we make.

2.2 Resource Allocation

Even though the approach taken in this thesis diverges significantly from that taken in mainstream computer science literature, we use tools which are similar to those used to solve other resource allocation problems. In this section, we review formulations used to solve resource allocation problems in a variety of other contexts. In Section 2.2.1, resource allocation strategies found in distributed computing systems are described, as a way of introducing the many different available formulations. To study the resource allocation problem when the service stations are not general purpose, Section 2.2.2 discusses the modeling and formulation issues related to resource allocation and scheduling of flexible manufacturing systems. Finally, due to the similarities between the routing problem in communications and the execution path selection problem in algorithm adaptation, approaches in data packet routing in communications networks are described in Section 2.2.3.

2.2.1 Distributed Computing Systems

In a distributed computing system, the resource allocation strategy manages the access to and the use of resource(s) by various consumers [11]. Depending on the context, the resources of a system may include entities from the physical processors and communications links, to software modules and peripherals, whereas the consumers include any process or task which requires the use of these resources. A "good" resource allocation strategy allows the consumers to quickly and efficiently access the resources, without incurring excessive overhead to execute the management function itself.

There are two major classes of strategies to assign processing tasks to processors: *static* and *dynamic*. In a static strategy, assignment decisions are made under the assumption that the set of processes currently in the system, as well as a complete description of all requirements of the given job, are known to the scheduler. Because of this assumption, a decision is possible before any part of the task is executed [40][29]. A dynamic strategy, on the other hand, relaxes the highly stringent

assumption that the resource needs of the given task are known a priori. Therefore, assignment decisions are made during execution, as the resource needs arise.

Dynamic strategies can be further classified by the location of the decision-making mechanisms. In a *physically distributed* strategy, the work involved in making allocation decisions is spread over multiple processors, whereas in a *physically non-distributed* strategy, all of the decision-making mechanisms reside in a single processor [44] [34]. The physically distributed strategies can be *cooperative* or *non-cooperative*. In cooperative strategies, even though many processors make allocation decisions, they all work toward a common system-wide goal [44]. In non-cooperative strategies, on the other hand, individual processors make decisions without regard for the effect of their decisions on the rest of the system [23].

There exist many tools for formulating the resource allocation problem and assigning processing tasks to processors. For instance, since there is a finite number of jobs and a finite number of processors, combinatorics can be used to enumerate the solution space and search for a solution which meets a set of desired conditions. In [41], for example, a state space search algorithm is used to find an optimal task-to-processor allocation when both execution and communications costs are accounted for. In devising the search strategy, it is crucial that the number of options generated and considered in the search space is kept to a minimum.

A more elegant class of formulations considers the resource allocation problem in the context of a network flow optimization problem such as maximum flow or graph coloring [46] [18]. For example, in [46], the author develops a graphical model of a modular program which can be used to determine the optimal partition of the program onto two processors. In this model, the nodes of the graph represent the modules and processors of the program. The arcs of the graph are associated with weights indicating the cost of running each module on each processor, as well as the cost of intermodule references when the modules are assigned to different computers. It is then proven that the optimal partition of the modules onto multiple processors can be solved by finding a cutset[1] in the graph with the least total weight [46]. In this way, the resource allocation problem has been reduced to a network flow problem for which there already exists many well-known algorithmic solutions.

Finally, a very common model useful for analysis of resource allocation policies is a queueing model in which each processor in the system is modeled as a service queue[16]. In [16], this type of model is used to analyze different policies for load balancing strategies in which tasks are dynamically transferred from busy to idle machines. Using known results from queueing theory as well as simulations, the paper shows that very simple load balancing policies offer dramatic improvements in performance, while in comparison, more complex policies which require more system information do not perform significantly better and are more susceptible to problems due

[1]A cutset of a graph is a set of arcs whose deletion would partition the nodes into disconnected subsets.

to their higher overhead and dependence on accurate data.

2.2.2 Flexible Manufacturing Systems

As explained earlier, in the distributed computing system literature, it is assumed that all processors are general purpose processors. There are many contexts in which specialized resources must be allocated among a wide range of different jobs. One such example is in manufacturing systems, where commonly each machining center is capable of performing only a subset of functions.

In a factory, products (more commonly called *part types*) are manufactured by passing raw material through a sequence of machines. In the simplest scenario, a factory consists of an assembly line of dedicated machines, each capable of performing only one operation. To smooth out variations in production rates, temporary storage is placed between machines, allowing machines to continue working while another machine undergoes repairs, tool changes, or maintenance. It is important to avoid starving or blocking a machine. A starved machine cannot produce because the preceding machine cannot supply it with parts. A blocked machine cannot produce because it has no place to send its finished parts.

In a more sophisticated scenario, the simple machines are replaced with machining centers capable of performing multiple operations. A parts transportation system is added to permit alternate routes which can compensate for machine failures. Such a manufacturing system is known as a *flexible manufacturing system*(FMS). Additionally, FMSs are often under the control of a common computer system which manages the flexibility afforded by the machines and transportation system.

Production planning in an FMS is the problem of deciding how much of each product should be produced at each time in an FMS, so as to meet a set of production demands on time [15]. This is a large and complex problem which involves large sets of decision variables on several different time scales. First, the machine data, part data, failure and repair rates, and demand information impose long term (days or weeks) constraints on the production capacity. Then, given these constraints, it is necessary to determine the short-term (hourly) production rates at which each product is produced at each moment of time. Finally, the scheduling of parts onto machines requires timing precision on the order of seconds or minutes. Clearly, scheduling for time scales of days or hours can be performed statically off-line, while scheduling with the precision of minutes or seconds may require dynamic adjustments.

Beyond the complexity of production planning, there are several key unavoidable modeling issues in the control of an FMS [15]. One of the most important is the uncertainty arising from random machine failures and repair rates, which impacts both short-term and long-term planning. In addition, there is often uncertainty in product demand due to forecasting inaccuracies and canceled orders. A third complication is that the manufacturing capacity of any factory is limited but unknown, and it is futile to schedule production at rates above capacity. Because of these

issues, the same factory may need to be analyzed via several different models, each suitable to study a limited set of characteristics. For example, Markov analysis may be used to analyze the effect of machine failures on the system, while queueing networks may be used to study the routing of parts throughout the factory.

In the current literature, the two major approaches to the problems of planning, control, and scheduling of FMSs are operations research and control theory. Works within the operations research and industrial engineering community have produced many analytical and planning tools using queueing models, mathematical programming models, and heuristic methods. Queueing models, first applied to the modeling of a FMS by Solberg [42], have been found to be useful for providing approximate guidelines for initial planning, but too unwieldy to be useful in operational planning [32]. To solve for actual scheduling of jobs on machines, mathematical programming models which solve constrained optimization problems are used instead. In these optimizations, typical objective functions include lateness, throughput, or average production time; the decision variables include production rates or inventory levels; the constraints account for production capacity and task priorities[15] [9]. Finally, due to the complexity of the planning and scheduling problems, heuristics have been used to ease the analysis and to increase the scope of applications of the mathematical formulations [30].

From the control theory perspective, the factory is a complex dynamic system whose behavior can be modified via control policies. Control theoretic methods have been particularly successful at determining short-term production rates, where it is possible to use aggregate models that relate part surplus levels to production rates and product demand [15]. In such formulations, the state variables represent the current surplus levels of the parts under production and the state of the machines, the input variables represent the product demands, and the control variables are the production rates of each part [22]. In simple cases, if a control policy which penalizes surpluses and backlogs is used, a *hedging point* policy is optimal. Such an operational policy sets the production trajectory to maintain a surplus level which is proportional to the amount of parts needed to satisfy the demand during a period of machine failures. The control theoretic approach to manufacturing has not yet matured. Concepts such as controllability and stability, which are well understood from a control perspective, have yet to be understood in the manufacturing context and remain open issues for further research.

2.2.3 Communications Networks

One of the most important services in a communications network is that of the routing of data packets from their origin to destination. As we discuss in Chapter 5, the routing problem is very similar to the problem of execution path selection in the context of algorithm adaptation. In both cases, packets(data blocks) may be transmitted(processed) through a variety of different

routes(execution paths). The goal of the routing strategy is to enable a large number of packets to move from origin to destination with low delay.

Like other resource allocation strategies, routing algorithms fall under many categories, ranging in levels of sophistication and efficiency. Particularly relevant to this thesis are distributed, adaptive algorithms, which allow the computation of routes to be shared among the network nodes, and adapt the routing decisions according to the current congestion levels in the network. There are two major classes of algorithms which satisfy these characteristics: shortest path routing algorithms and optimal routing algorithms.

In shortest path routing, each communication link is assigned a length which may depend on its transmission capacity and projected traffic load. Then, given these lengths, the shortest path between any two nodes can be calculated efficiently using well-known algorithms such as the Bellman-Ford algorithm, Dijkstra's algorithm, and the Floyd-Warshall algorithm [6]. Even though it is in use in many modern networks today, shortest path routing suffers from two major shortcomings. First, because there is only one shortest path between any origin-destination pair, all packets between those two network nodes are placed on the same path, thereby limiting the potential throughput between those two nodes. The second problem is that the algorithm may lead to instabilities and oscillations in the network. Since the link traffic levels depend on the routing, and the routing depends on the link traffic levels, the feedback can cause oscillatory routing behavior. In practice, these oscillations are dampened by making the link length estimates less sensitive to the link arrival rate, but this approach reduces the algorithm's sensitivity to congestion.

Optimal routing eliminates the problems of shortest-path routing by optimization of an *average* measure of performance [6]. In this approach, the routing problem is formulated as a nonlinear constrained optimization problem in which the average system congestion is minimized subject to the constraint that the system handles all of the incoming traffic. A characterization of this optimization problem shows how data packets between the same origin-destination node pair should be simultaneously placed on multiple routes in order to balance the congestion level throughout the system. Many different computational procedures are available to solve the optimization problem dynamically on a network. Most interesting among them is the gradient projection method, which is an iterative algorithm that calculates the flow change increments based on relative magnitudes of the path lengths and, sometimes, the second derivatives of the cost function. The main limitation associated with optimal routing methods is that the cost function generally depends only on the first-order (average) statistics of link traffic. Undesirable behavior associated with high variance and with correlations of packet interarrival times and transmission times is thereby ignored. Nevertheless, it is effective and achieves good stable performance in most cases.

2.3 Summary

The goals of this chapter are twofold. First, even though the subject of this thesis contains some similarities with the field of distributed computing, it presents a significantly different perspective of the problems, issues, and opportunities associated with such systems. It is the primary purpose of this chapter to delineate these differences. The second goal of the chapter is to aid the development of the thesis by surveying tools for solving resource allocation problems. Therefore, an introduction to formulations of resource allocation problems in a variety of domains is presented in Section 2.2. Our discussion clearly reveals that there are numerous approaches offering different tradeoffs in terms of expected performance, implementational complexity, and analytical complexity.

Chapter 3

System Model

As stated in the introduction, this thesis is concerned with DSP on dynamic heterogeneous computing networks. As such, a fundamental component of the study is the development of a tractable and analytical model for such systems. In this chapter, we begin by presenting a detailed description of the class of hardware environments under study in this thesis. As is evident from our discussions in Chapter 2, there are many ways to model systems of processors. In Section 3.2, we present a system model which we find to be particularly tractable and suitable for the purpose of algorithm adaptation. The system description and model are then further clarified through examples of such environments in Section 3.3.

This chapter sets the stage for the remainder of the study in several ways. First, by defining and describing the class of systems under study, we hope to make clear the explicit assumptions and requirements of this work. Second, the development of the system model permits us a simple foundation on which the remainder of the thesis can be built. Third, by considering some concrete examples of the class of systems under study, we motivate our work and show its relevance to current and future computing systems.

3.1 System Description

The three major characteristics of the assumed computation environment are heterogeneity, time-variance, and specialization on signal processing. The explicit consideration of heterogeneous dynamic systems sets this work in a highly variable environment in which a user has little knowledge of the computation architecture prior to actual execution of his algorithm. The focus on signal processing in turns restricts our attention to a set of highly mathematical, highly structured algorithms.

3.1.1 Heterogeneity

The heterogeneity of the network is a characteristic of its devices and links, referring to the property that different devices have different limitations in their power, memory, speed, etc. For example, some processors can be tiny battery-operated microprocessors, while others can be powerful supercomputers; similarly, the links can range from low rate wireless channels to high speed fiber-optic lines.

From the point of view of a user, the heterogeneity of the environment complicates his choice of algorithms. The reason is that one's choice of algorithms greatly depends on one's prediction of the "bottleneck" of the computational platform. For example, while time and memory usage are the criteria of interest for many microprocessor platforms, the power consumption of an algorithm is the optimization criterion of choice in low-power systems. An algorithm which is highly efficient on a certain type of processor may be very undesirable on another type. Finally, the bottleneck of the system may not reflect that of any individual processor. For example, the bottleneck of a large farm of processors may not be processing complexity, even if each of the processors is limited in processing capability.

From the point of view of the system designer, variations in limitations of capability imply that not all processors can be used for all processing jobs. As an extreme example, in an environment composed of Application-Specific Integrated Circuit (ASIC) chips, each processor implements only one algorithm. As a result, the heterogeneity of the environment may complicate the considerations required to assign jobs to processors. For instance, in a homogeneous environment, the system can follow relatively simple rules, assigning jobs to the nearest idle processor as they arrive, or migrating jobs to balance the load throughout the network. In a heterogeneous environment, on the other hand, the system needs to factor in the processing and communications capacity of each processor and link.

3.1.2 Time Variance

The dynamic nature of the network refers to the time-variance of its configuration or state. In this thesis, the state description of a processor goes beyond a simple on/off description to include a variety of properties:

- The *reachability* of a processor with respect to the network refers to whether a processor is utilizable to the network. It may vary over time as processors move around in the network, as the communications links fail or recover, and for reasons such as security considerations.
- The *functionality* refers to the type of algorithms which can be executed by the processor. It varies over time for programmable processors such as Field-Programmable Gate Arrays(FPGAs), whose functionalities change as different hardware or software configurations

are loaded.

- The processing *capacity* refers to the maximum throughput of the processor given its current functionality. This property may change over time as a reaction to physical circumstances. For instance, a processor may scale back its operations if it senses over-heating, or if its battery power level drops. More drastically, the capacity drops to zero when the processor fails.

- Finally, the *current load* of any processor in the system is subject to two major influences: the user demand and the resource allocation policy. The variation in user demand levels is a major source of time variance of the system state. The resource allocation policy, while not necessarily time-varying, plays a paramount role in dictating the distribution of load throughout the system.

Analogously, the state of a communications channel is represented by two quantities: the load and the capacity. Similar to processor loads, the load on a communications link depends on user demand and the resource allocation policy. The channel capacity may change over time for a few reasons. First, for point-to-point channels over wire or fiber, the capacity can drop to zero if the transmission line is cut. At the same time, as the level of interference experienced on a wireless link varies over time, so does the capacity of the link. Furthermore, in multi-access communications, the capacity available to each user of the channel depends on the activity of the other users.

To the system user, a time-varying computing environment is an additional layer of unpredictable behavior which makes algorithm optimization even more difficult, since the type of processor suitable for a chosen algorithm may not even be available at the time of processing. To the system designer, the dynamic behavior of the computing environment motivates an adaptive resource allocation strategy, so that the system can behave efficiently over a wide range of system states. The idea of algorithm adaptation, to be described in detail in Chapter 5, is developed in this thesis to address the concerns of both the user and the system designer. With this technique, the choice of the actual algorithm used to complete a processing task is made by the system (rather than the user) according to its current state.

3.1.3 DSP Focus

Finally, the third major system characteristic is the system's specialization on signal processing. It can be considered as a reflection of our focus on signal processing, rather than a literal restriction of the type of system under study. Our focus represents two considerations. First, signal processing algorithms represent a rich, interesting, and often computationally intensive subset of all algorithms. Second, as will be seen in Chapter 4, this specialization allows us to take advantage of the inherent structure in DSP algorithms to formulate a dynamic algorithm optimization framework. As a result,

for the purpose of the study, system capabilities beyond those required to implement specific signal processing algorithms will not be considered during system modeling, design, or evaluation.

3.2 System Model

As with any computation system, the design of a distributed signal processing system involves a variety of questions ranging from protocol design to communications issues. In this thesis, rather than studying protocol issues, we concentrate on adaptation issues arising from the unreliable and dynamic nature of the system. Specifically, the technique of algorithm adaptation, in which the algorithm used to complete a processing task is allowed to change according to the state of the system, is investigated. To support this research direction, the system model has two parts. The first part captures sufficient information to facilitate the algorithm adaptation strategy, while the second hides the details in those issues which will not be addressed, such as protocol and communications.

3.2.1 Collection of Primitive Operators

To consider the information necessary for algorithm adaptation, we first explain the basic intuition behind it, even though more details are presented in Chapter 5. Based on our discussion of system states earlier, it should be clear that while some of the sources of variability are beyond the control of the engineer, others can be manipulated to influence the state of the system. For example, the load on the system can be affected in at least two ways. Clearly the resource allocation policy is a crucial part of the system design. Additionally, the choice of algorithms by the user changes the user demand, and hence the load, on the system. Therefore, the design of bandwidth and computationally efficient algorithms is a fruitful and common way to improve the performance of the system. Algorithm adaptation can be considered as a hybrid strategy in which the resource allocation strategy involves dynamically choosing among a set of functionally "equivalent" algorithms.

For the system to choose one algorithm over another, it needs some ability to predict how the performance of the system will change based on its choice. More specifically, the system should consider how its state will change as a result of its decisions. To facilitate this process, our system model should allow us to easily encapsulate the knowledge of the current state of the processors, namely the reachability, capacity, functionality, and load of each processor, regardless of its type.

In this work, we model the processing network as a collection of specialized virtual processors whose functions are elements of a set of "primitive" operations with which DSP algorithms are composed. Some examples of these primitives may include radix-2 FFT, convolution, and sequence addition. The functional descriptions may also include implementation limitations such as finite word length implementations or minimum transition widths for bandpass filters. The physical

manifestation of each virtual processor is unspecified. Hence a virtual processor may reside on one or multiple physical processors, and each physical processor may behave as several virtual processors. The function of any particular physical processor may also change. This is modeled by assigning multiple virtual processors to reside within the same physical processor, but allowing only one of them to operate at a time. Each virtual processor is also referred to as a *network node*. Statistics encapsulating the reachability, capacity, and load of each network node form a mathematical expression of the system state.

Since they are the building blocks of the algorithms implementable in the system, the design of the set of primitive operations is important. First, the set of primitive functions provided in the system must be sufficiently complete so that a large number of algorithms can be specified using these basic functions. Secondly, the granularity of this set of functions must be appropriate. If all of the primitives perform high-level functions, alternative implementations of algorithms become difficult to specify, resulting in fewer opportunities for adaptation. In the other extreme, if the primitives perform only bit-wise multiplication or addition, algorithm specifications become impossibly cumbersome, and the cost of communications and control overhead incurred from passing the data from point to point in the network dominate the processing costs. A set of network nodes is said to have "full functionality" if their functions span the complete primitive function set.

This aspect of the model reduces the processor network to a simple representation of a collection of virtual processors, each with a known functionality and capacity. It applies equally well to general purpose multi-processors as well as ASICs. With respect to algorithm adaptation, it allows the system to consider algorithms as the compositions of the basic primitive operations on the network nodes. Therefore, the choice between two algorithms involves choosing among the two respective sets of virtual processors used in the algorithm compositions. Details on the physical processors are hidden to allow us a tractable view of the relationship between the algorithm specification and the physical processors.

3.2.2 Object-Oriented Data Blocks

To hide the protocol details, the second part of the model involves an object-oriented model of the data which require processing. Specifically, a data object is a data block tagged with a description of the required processing steps; the system routes the data block to processors according to these specifications. Processors maintain processing queues to store those data packets requiring its work. As the data is transformed by each processor, the tagged description changes accordingly to reflect the state of the data. Part of our research deals with how algorithms can be specified to give the system the most flexibility in adapting them to the system conditions. Using this model allows us concentrate on the dynamic selection of processing steps without worrying about the details of inter-processor protocols.

To hide the communications details, we assume that the underlying network is capable of routing data packets to the desired processing location with reasonable delay. While communications links and network nodes may fail and recover randomly, we assume that no catastrophic failure occurs. In other words, every working node is connected to a set of working network nodes with full functionality at all times. Furthermore, we assume that data is not lost due to the failures of the links and processors. This can be accomplished through failure recovery mechanisms within the communications networks itself. For simplicity, communications, control, and memory costs are not explicitly considered in this study. As our understanding and techniques mature, these considerations will become increasingly important. At the present time, however, we assume that by choosing reasonably high-level primitives and reasonably long data blocks, these costs will not be prohibitively high.

3.3 Examples of Computing Environments

Thus far in this chapter, we have described and modeled a class of computing environments without detailing the specifics. In this section, we describe serveral specific computing environments and operating scenarios which fall into the class of dynamic heterogeneous DSP environments studied in this thesis. Hopefully, these examples will further motivate this work by illustrating its applicability to present and future computing systems.

3.3.1 Internet

The Internet is an excellent example of a heterogeneous and dynamic network. Any processor implementing the TCP/IP protocols can communicate through it, and links and processors can connect and disconnect at any time without affecting the overall operation of the network. With the robustness and strength of its communications protocol design, it has grown rapidly as a mean of sharing information in the recent years. Its potential for sharing computational power, however, remains largely untapped.

Consider the following scenario on the Internet. A user with a piece of data requiring processing encloses it in a special file format, along with processing instructions. He sends the file over the Internet, where the enclosed data is processed according to the instructions by a sequence of processors on the way. The network then sends the final result back to the user.

However, the composition of the internet is dominated by general purpose microprocessors. Why would it be reasonable to model it as a collection of virtual specialized processors? First, it may be impossible to specify the processing instructions in low-level compiled code, due to incompatibilities among different processors. Second, for the purpose of efficient specification of algorithm choices, it may be more efficient to use the primitive operations in our system model as the building blocks

of the processing instructions. In this scenario, because machine-level code is not transmitted, the functionality of a given microprocessor depends on its ability to implement a given primitive operation, or more specifically, the availability of machine-level code implementing the primitive operation at the processor. With a large variety of primitive operations, it is likely that processors will not store all of the possible codes locally. Instead, it may be convenient to imagine repositories of machine-level code within the Internet where individual microprocessors can download functionalities as needed. In essence, then, each individual general purpose microprocessor is actually a composite of a finite number of virtual processors whose functionalities are available in machine code form locally.

In such scenarios, consider the system's decision process with regard to the processing instructions attached to the data packet. Based on its knowledge of the "distribution" of the virtual processors (which is likely to be time-varying), it must choose an execution path incurring the minimum delay or system cost. At the same time, the system may want to consider the reverse problem: based on its knowledge of the processing demands of the users, decide which primitive operations each microprocessor should keep locally to result in the best performance.

3.3.2 DSP Network Chips

In the previous example, we assume that the communications and processing capabilities of the processors are entirely separate. As semiconductor technology advances, however, it has brought about new chips with both networking and DSP capabilities, making powerful DSP chips with embedded network functions which can serve as the switches and routers inside networks. Even as they perform the network functions, these DSP network chips contain spare capacity which can be utilized for additional processing.

Consider a network in which a subset of routers are implemented with such chips, each of which is programmed with a primitive DSP function. As data packets are transmitted across such networks, their processing and communication requirements can be satisfied simultaneously if they travel through an appropriate sequence of the DSP-enabled routers. Such a network is heterogeneous, for not all routers are DSP-enabled, and not all DSP-enabled routers are programmed to perform the same function. It is dynamic in the usual sense that links and network nodes may fail unpredictably. The system model described in Section 3.2 clearly models its behavior appropriately.

In such systems, the routing of the data packets involves considerations of minimizing the total of the processing and communication time. At the same time, the system designer would need to consider how to periodically update the functionalities of the DSP processors to fit the user demands.

3.3.3 FPGA Networks

Field-programmable Gate Arrays (FPGAs) are on-board programmable chips capable of implementing complex register-intensive logic functions. Compared with general purpose processors, FPGAs are capable of higher performance at lower cost; compared with custom built chips such as ASICs, FPGAs offer faster development time with less risk [49]. Moreover, libraries of gate array layouts implementing DSP functions such FIR filters, correlators, and FFTs already exist [49].

Networks composed of FPGAs devices exemplify the type of distributed computing envirionments under study in this thesis. The primitive building blocks are naturally those functions contained in the DSP libraries. Because each FPGA in the network may implement a different function, the environment is clearly heterogeneous and can be modeled as a collection of virtual specialized processors. Additionally, because each chip is on-board programmable, the exact composition of the overall network is likely to be time-varying. The system model described in this thesis easily captures the characteristics of such networks.

3.4 Summary

In the class of computational environments under study in this thesis, the processors are heterogeneous and time-varying in both functionality and capacity. In this chapter, we develop a model of these environments as networks of mutually connected specialized virtual processors. This system model sufficiently captures the salient system state characteristics for the purpose of algorithm adaptation. Additionally, details in communications and other control protocols are hidden through the encapsulation of data with a description of the required processing task into data block objects, allowing us to concentrate on issues beyond protocol. The overall model deviates significantly from other system models used in study of distributed computing systems, because it does not assume that every processor is a general purpose microprocessor.

The systems described in Section 3.3 serve as simple examples of environments built with devices available today. We believe that as different processor architectures emerge, highly heterogeneous and dynamic network environments of the type under study will become even more prevalent. This trend is an important factor in evaluating the applicability of this thesis to future systems.

Chapter 4

Algorithm Specification

The development of algorithm adaptation requires two major building blocks. The first is the system model developed in Chapter 3, which encapsulates the processor state information while hiding protocol and communications details. The second is the algorithm specification language to be presented in this chapter, which expresses the desired processing objectives in terms of a variety of possible implementations. Combined, these two components allow the system to dynamically evaluate and choose among the various implementations, making algorithm adaptation possible.

This chapter begins the development of the algorithm specification language with the definition of *functional equivalence* among algorithms in Section 4.1. Section 4.2 then presents the details of the algorithm description language through which functionally equivalent algorithms can be efficiently expressed. A graphical representation of this language results in a graphical model of algorithms which is described in Section 4.3 and interpreted in Section 4.4. Throughout this chapter, the definitions and constructions are exemplified with a simple but interesting set of spectral analysis algorithms.

4.1 Functional Equivalence of Algorithms

In this thesis, two algorithms are functionally equivalent with respect to a given data set if the qualities of their outputs are equally acceptable to the user. This definition coincides with statements of processing objectives of the form: "Given the data and its known characteristics, process the data to return the desired information with quality measure of at least x." Depending on the type of the algorithm, the quality measure may refer to factors such as probability of error, compression ratio, or signal-to-noise ratio. In particular, functionally equivalent algorithms do not necessarily produce the same results even when given the same data set.

Signal processing algorithms can be functionally equivalent to others through several means. On the most direct level, equivalence can be established simply by rearrangements of the mathematical

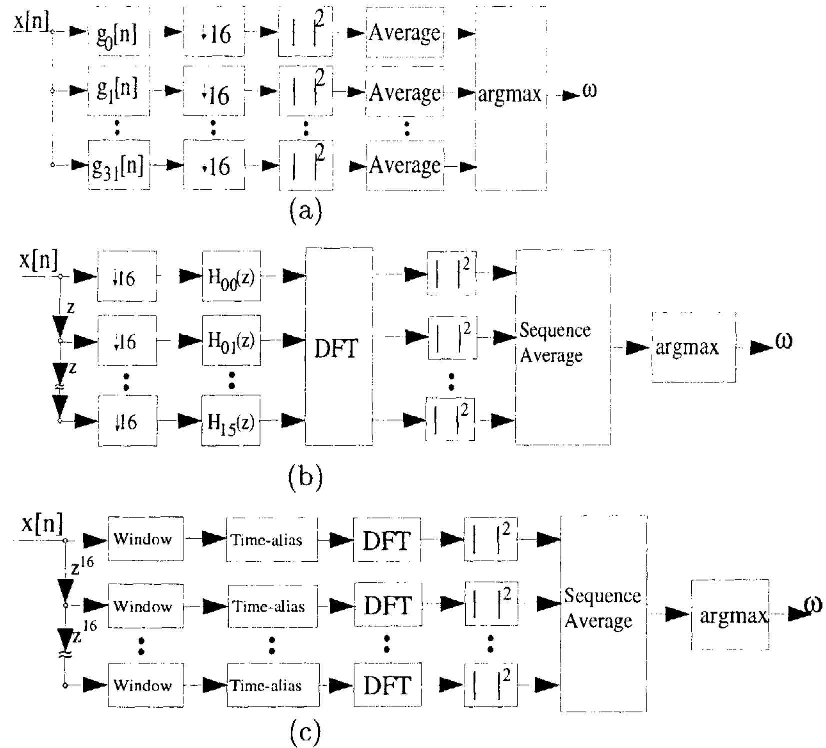


Figure 4-1: Block diagrams of 3 functionally equivalent algorithms for periodogram averaging to find the peak frequency of a spectrum. (a) Modulated filter bank. (b) A polyphase implementation of the filter bank. (c) Short-Time Fourier analysis implementation.

operations which compose the algorithm. Such equivalent algorithms would produce identical results for all data sequences. Some common examples are

- Convolution in the time domain & multiplication in the frequency domain.
- High-order filters expressed as a cascade of lower-order filters & the same filter expressed as a sum of parallel lower-order filters.
- Filtering operations preceded or followed by up- or down-samplers & the corresponding polyphase decomposition implementation.

Example. A richer example of this type of functional equivalence involves the three algorithms shown in Figure 4-1. All three algorithms find the peak frequency of a spectrum via periodogram averaging. They produce exactly the same output for every input sequence, and are derivable from each others based on rearrangements possible through polyphase structures of the filters[48].

In the algorithm depicted in Figure 4-1(a) the signal is first passed through a modulated filter bank. The filter outputs are downsampled, and then the average is taken of the magnitudes of the sequences. The location of the peak in the spectrum is found by choosing the channel with

the highest average magnitude. Figure 4-1(b) shows a polyphase implementation of the modulated filter bank. The signal is separated into its polyphase components, and each of those are filtered by the polyphase decomposition of the prototype filter in the modulated filter bank. DFTs are taken of the outputs at each time frame, and the location of the peak is found by finding the highest point in the average of the squared-magnitude of those spectra. The implementation in Figure 4-1(c) structures the computation as a short-time Fourier transform. The signal is windowed in time, and each of the windowed segments is appropriately time-aliased and Fourier transformed. The last sections of this implementation are the same as in the polyphase implementation, so again, the location of the peak is found by finding the highest point in the average of the squared-magnitude of those spectra. □

More generally, however, functional equivalences are due to algorithmic development work which resulted in classes of algorithms offering different types of tradeoffs. For example, the literature describes a variety of lossless compression algorithms which achieve different compression ratios while incurring different processing costs. In this case, all algorithms achieving or exceeding the given objective compression ratio are functionally equivalent.

The functional equivalence among algorithms may further depend on the characteristics of the input data. As a simple example, in channel equalization for a time-invariant communications channel, adaptive and non-adaptive algorithms are functionally equivalent to each other; such would not be true for a time-varying channel. A more complicated example involves the set of image formation algorithms for stripmap synthetic aperture radar (SAR) to be described in Chapter 8. As shown there, this class of algorithms trades off computational complexity for efficacy against increasing amounts of imaging distortions. Hence, a data set collected with only mild distortions can be processed using any of these algorithms, while a highly distorted data set is restricted to using only the more powerful among the set.

Functional equivalence exists frequently among signal processing algorithms due to the highly mathematical nature of both their formulations and their specifications. This characteristic of DSP algorithms makes algorithm adaptation possible and attractive. While the design of sets of functionally equivalent algorithms is in itself a rich and interesting problem, we leave its exploration to future work. Rather, we take the point of view of the system designer, and assume that the user is able to recognize, select, or design sets of functionally equivalent algorithms, and present such sets to the system. The problem is then how the system should choose among these available choices to improve its performance.

4.2 Algorithm Description Language

To take advantage of the availability of functional equivalence among DSP algorithms, we must be able to efficiently describe sets of functionally equivalent algorithms to the system. In a transparent distributed system, in which the user submits the same processing commands regardless of system state, such descriptions make up the interface between the user and the system. This section specifies this interface by introducing an algorithm description language whose design is guided by two major considerations. First, since the computation environment consists of a large set of processors, it can support a great deal of parallelism, and hence the expressivity of concurrency is important. Second, the language must efficiently express functional equivalences between algorithms, highlighting their similarities and differences. By attaching such descriptions to each data block, the user leaves the system to choose among the alternatives at runtime, with the most updated information available.

4.2.1 Notation

The chosen description language has a notation similar to traditional operator notation, with a special meaning for the plus sign. In fact, due to this notational similarity, the resulting algorithm descriptions are also called "algebraic" expressions.

An algorithmic expression consists of two parts. The first part is an algebraic composition of primitive operators expressing the operations on the data, while second part is either a vector or a singleton symbol representing the data. In the thesis, primitive operators *and* their compositions are typeset in the same way in `Capitalized Typetext`, because they both represent operations on data. On the other hand, symbols for their operands, the data blocks, are typeset in `lowercase typetext`.

The description language allows the following three methods of composition:

- Sequential: The expression `y = BAx` means that the data represented by `x` is operated on first by `A` and then by `B`.
- Concurrent: Concurrent operations are written as element-wise multiplication.

$$\begin{bmatrix} \mathtt{y}_1 \\ \mathtt{y}_2 \end{bmatrix} = \begin{bmatrix} \mathtt{A} \\ \mathtt{B} \end{bmatrix} .* \begin{bmatrix} \mathtt{x}_1 \\ \mathtt{x}_2 \end{bmatrix}$$

  means that `A` operates on $\mathtt{x}_1$ while `B` operates on $\mathtt{x}_2$.
- Equivalent: Equivalences among operations are separated by a `+` sign. `y = (BA+AB)x` implies that operations `A` and `B` are commutative.

Since this language uses matrix-like notation, the correct dimensionality must be maintained. The

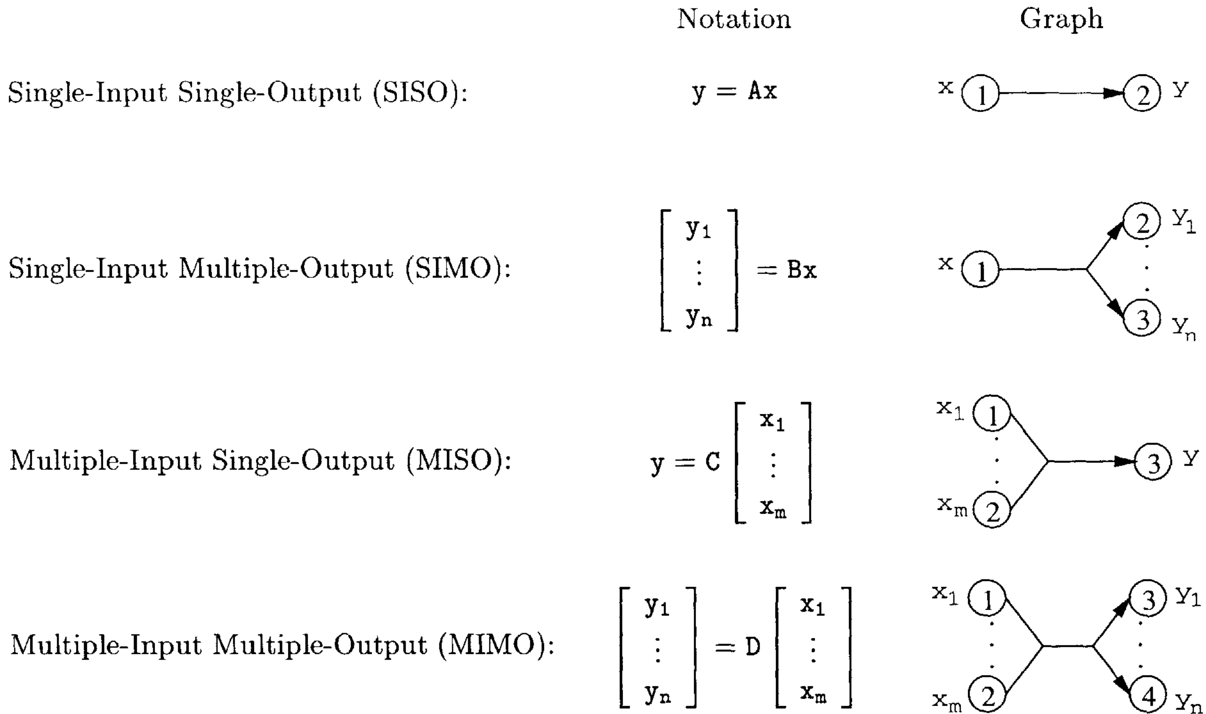


Figure 4-2: Notation and Arc shape for single-input single-output, single-input multiple-output, multiple-input single-output, and multiple-input multiple-output functions

left column of Figure 4-2 shows the correct notation for primitive operators with multiple inputs and/or outputs.

The language also defines two special classes of n-input n-output operators `Identity` and `Permutation`. The class of `Identity` operators contains operators of varying dimensionality whose effect on data blocks is as the name suggests. Given a vector of n data blocks, the `Identity-n` operator leaves the vector unchanged. The class of `Permutation` operators re-orders its input vector of data blocks according to a permutation table. For a given dimension n, there are $n!$ different n-input n-output `Permutation` operators.

Finally, the following rules apply:

- Commutativity of Equivalences: $(\mathtt{A} + \mathtt{B})\mathtt{x} = (\mathtt{B} + \mathtt{A})\mathtt{x}$
- Identity: $(\mathtt{Identity\ A})\mathtt{x} = (\mathtt{A\ Identity})\mathtt{x} = \mathtt{Ax}$ for the `Identity` operator of the correct dimension.
- Distributive Law 1: $(\mathtt{A} + \mathtt{B})\mathtt{Cx} = (\mathtt{AC} + \mathtt{BC})\mathtt{x}$
- Distributive Law 2: $\mathtt{A}(\mathtt{B} + \mathtt{C})\mathtt{x} = (\mathtt{AB} + \mathtt{AC})\mathtt{x}$.
- Distributive Law 3: $\begin{bmatrix} \mathtt{A} + \mathtt{B} \\ \mathtt{C} + \mathtt{D} \end{bmatrix} .* \begin{bmatrix} \mathtt{x}_1 \\ \mathtt{x}_2 \end{bmatrix} = \left(\begin{bmatrix} \mathtt{A} \\ \mathtt{C} \end{bmatrix} + \begin{bmatrix} \mathtt{A} \\ \mathtt{D} \end{bmatrix} + \begin{bmatrix} \mathtt{B} \\ \mathtt{C} \end{bmatrix} + \begin{bmatrix} \mathtt{B} \\ \mathtt{D} \end{bmatrix} \right) .* \begin{bmatrix} \mathtt{x}_1 \\ \mathtt{x}_2 \end{bmatrix}$

- Distributive Law 4: $\begin{bmatrix} \texttt{AC} \\ \texttt{BD} \end{bmatrix} .* \begin{bmatrix} \texttt{x}_1 \\ \texttt{x}_2 \end{bmatrix} = \left(\begin{bmatrix} \texttt{A} \\ \texttt{B} \end{bmatrix} .* \begin{bmatrix} \texttt{C} \\ \texttt{D} \end{bmatrix} \right) .* \begin{bmatrix} \texttt{x}_1 \\ \texttt{x}_2 \end{bmatrix}$, if the number of outputs of C equals the number of inputs of A; and the number of outputs of D equals the number of inputs of B;

Example. Continuing the example from the previous section, we now present the algorithmic description of the three algorithms for spectral analysis (shown in Figure 4-1). To begin, assume that the primitive functions shown in Figure 4-3 are available in the system. Figure 4-4 shows the algebraic expressions specifying the three functionally equivalent algorithms, using the abbreviations defined in Figure 4-3. The first line of the expression corresponds to the modulated filter bank algorithm in Figure 4-1(a). This line incorporates the equivalence between the operations of convolution (`Cnv`) and FFT-Multiply-IFFT (`IftMulFft`). Additionally, the operations of downsampling (`Dwn`) a sequence and finding the magnitude(`Mag`) of its samples are commutative, giving rise to the expression `(DwnMag + MagDwn)`. The second and third lines correspond to the polyphase and the short-time Fourier analysis algorithms of Figure 4-1(b) and (c). Since the second halves of these two algorithms are identical, the second line corresponds to the latter half of both, while the third line highlights their difference. The polyphase algorithm requires a Serial-to-Parallel converter (`Srp`) followed by a bank of filters (`Cnv + IftMulFft`), while the short-time Fourier analysis algorithm requires windowing (`Rct`) followed by time-aliasing (`Tma`). □

4.2.2 Concurrency Operations

Even though element-wise multiplication is a simple notation for concurrent operations, some issues regarding the usage of multi-input or multi-output operators within these structures deserve discussion.

In concurrency expressions with multi-input or multi-output operators, the matching between operator and operand may not be clear. In such cases, the expression can be clarified using block

Functionality	Abbr.	Functionality	Abbr.
Argmax	`Arg`	FFT	`Fft`
Average	`Avg`	IFFT	`Ift`
Average Sequence	`Avs`	Magnitude	`Mag`
Autocorrelation	`Aut`	Multiplication	`Mul`
Convolution	`Cnv`	Rectangular Window	`Rct`
Downsample	`Dwn`	Serial-Parallel	`Srp`
Replicate	`Rep`	Time-Aliasing	`Tma`

Figure 4-3: List of system primitive functions and abbreviations

$$
\begin{aligned}
\mathtt{y} \;=\; & \left(\left\{\mathtt{Arg}\begin{bmatrix}\mathtt{Avg(DwnMag+MagDwn)(Cnv+IftMulFft)}\\ \vdots \\ \mathtt{Avg(DwnMag+MagDwn)(Cnv+IftMulFft)}\end{bmatrix}\mathtt{Rep}\right\}\right.\\
& +\left\{\mathtt{ArgAvs}\begin{bmatrix}\mathtt{Mag}\\ \vdots\\ \mathtt{Mag}\end{bmatrix}.*\begin{bmatrix}\mathtt{Fft}\\ \vdots\\ \mathtt{Fft}\end{bmatrix}.*\right.\\
& \left.\left.(\begin{bmatrix}\mathtt{Cnv+IftMulFft}\\ \vdots\\ \mathtt{Cnv+IftMulFft}\end{bmatrix}\mathtt{Srp}+\begin{bmatrix}\mathtt{Tma}\\ \vdots\\ \mathtt{Tma}\end{bmatrix}\mathtt{Rct})\right\}\right)\mathtt{x}
\end{aligned}
$$

Figure 4-4: Algorithm description of the spectral analysis algorithms in Figure 4-1.

matrix notation. For example, the following expression adds $\mathtt{x}_1$ and $\mathtt{x}_2$ while squaring $\mathtt{x}_3$:

$$
\begin{bmatrix}\mathtt{y}_1\\ \mathtt{y}_2\end{bmatrix}=\begin{bmatrix}\text{add}\\ \\ \text{square}\end{bmatrix}.*\begin{bmatrix}\begin{bmatrix}\mathtt{x}_1\\ \mathtt{x}_2\end{bmatrix}\\ \mathtt{x}_3\end{bmatrix} \tag{4.1}
$$

Additionally, in concurrency expressions, the length of the operator vector and the number of blocks in the operand vector must be equal. As a result, the language construction excludes the possibility that two operators act on the *same* data block simultaneously. For example, the following expression is invalid because the operator vector has 2 elements, while the operand is a single data block:

$$
\begin{bmatrix}\mathtt{y}_1\\ \mathtt{y}_2\end{bmatrix}=\begin{bmatrix}\sin\\ \cos\end{bmatrix}.*\mathtt{x}
$$

The correct expression is

$$
\begin{bmatrix}\mathtt{y}_1\\ \mathtt{y}_2\end{bmatrix}=\left(\begin{bmatrix}\sin\\ \cos\end{bmatrix}.*\mathtt{Replicate\text{-}2}\right)\mathtt{x}
$$

The expression of simultaneous operations on the same data requires the explicit instruction that the data block is first replicated, with the operations performed on the copies. While this consequence of the requirement for the vector dimensions to match may seem cumbersome, it correctly reflects the way the system handles a request to process the same data block through multiple operators; each processor works on its own copy of the data.

A third issue is that a given vector of data blocks may not be properly ordered for a vector of concurrent operators to be applied. For instance, suppose that given the data block vector $[\mathtt{x}_1\ \mathtt{x}_2\ \mathtt{x}_3]$,

one wishes to add x_1 and x_3 while squaring x_2. In this case, the operator vector should be preceded with a `Permutation` operators to re-order the variables into an appropriate arrangement. In the following, for example, `Permutation-132` re-orders $[x_1\ x_2\ x_3]$ to $[x_1\ x_3\ x_2]$:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \left(\begin{bmatrix} \texttt{add} \\ \\ \texttt{square} \end{bmatrix} .*\texttt{Permutation-132} \right) \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{4.2}$$

Finally, in the event that the concurrent operations include only some of the data blocks, the `Identity` operator is necessary as a place-holder. For instance, suppose that given the data block vector $[x_1 x_2 x_3]$, one wishes to sum the square of x_1, the cube of x_3, and x_2, the following expression is appropriate:

$$y = \left(\texttt{sum} \begin{bmatrix} \texttt{square} \\ \texttt{Identity} \\ \texttt{cube} \end{bmatrix} \right) .* \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

4.2.3 Discussion

Two other characteristics of the description language must be noted. First, the design of the system model imposes a single restriction on the algorithmic expressions attached to the data blocks on the system. Namely, the expressions may use only those primitive operations specified in the system model. This makes it possible for the system to analyze the attractiveness of each option in terms of its available statistics. The restriction highlights the importance of the design of the set of primitive operations in the system. As discussed in Chapter 3, proper considerations for the completeness and the granularity level of the primitive set must be taken.

Second, while the language is well-fitted to sequential and parallel operations, it does not include looping and conditional constructs. These constructs are not included in the language for now so that we can better focus on the portion of control which would benefit most from algorithm adaptation. Nevertheless, the inclusion of these constructs represents an interesting and significant extension of this work.

4.3 Algorithm Graph

The algebraic expressions can also be represented graphically in structures called algorithm graphs. As we will see, algorithm graphs are interesting structures for studying a variety of issues related to distributed signal processing. This section introduces the construction, nomenclature, and assumptions associated with such graphs.

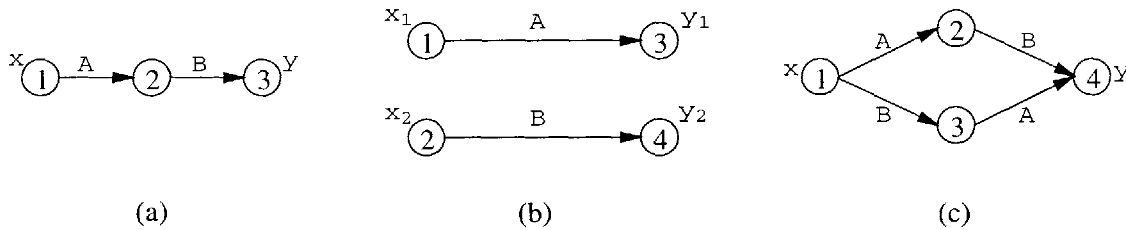


Figure 4-5: Algorithm graph constructions for (a) sequential operations; (b) concurrent operations; (c) equivalent operations.

4.3.1 Construction

Algorithm graphs are directed acyclic graphs whose nodes represent states of the data and whose arcs represent operations on the data. The construction of an algorithm graph from an algebraic expression is quite simple. The three constructs of the language are graphically represented in Figure 4-5. Notice from Figure 4-5 that every node of an algorithm graph is associated with a node number. If a node corresponds to a named variable in the algebraic expression, it is labeled outside of the circle.

A sequence of operators is depicted by simply stringing arcs in a row, as shown in Figure 4-5(a), which corresponds to the algebraic expression $\mathtt{y} = \mathtt{BAx}$. Figure 4-5(b) shows the construct for concurrent operations, and corresponds to the algebraic expression

$$\begin{bmatrix} \mathtt{y}_1 \\ \mathtt{y}_2 \end{bmatrix} = \begin{bmatrix} \mathtt{A} \\ \mathtt{B} \end{bmatrix} .* \begin{bmatrix} \mathtt{x}_1 \\ \mathtt{x}_2 \end{bmatrix}$$

Finally, Figure 4-5(c) corresponds to the algebraic expression $\mathtt{y} = (\mathtt{BA} + \mathtt{AB})\mathtt{x}$. It shows that multiple outgoing arcs originating from a single node represent the fact that multiple functionally equivalent algorithms are available for processing the data block at that node. Hence, such nodes represent the decision points for the system, and multiple paths between nodes imply that there are multiple methods for accomplishing the same processing objectives.

Since operators can take single or multiple inputs and result in single or multiple outputs, there are four distinct types of arcs, shown in the right column of Figure 4-2. The arcs have single or multiple head and tail nodes corresponding to the number of inputs or outputs in the processing function they represent. In this way, each operator in the algebraic expression is translated into an arc in the algorithm graph. This rule applies to all operators except the `Identity` and `Permutation` operators. The `Identity` operators do not change the data in any way, and therefore do not need to be explicitly drawn. A `Permutation` operator alters only the order of its input vector so that subsequent operations act on the desired set of inputs. Graphically, its effect can be accounted for

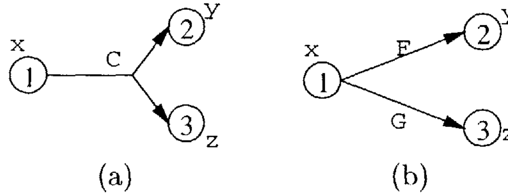


Figure 4-6: (a) The data block x is transformed by the single-input dual-output primitive C. (b) The data block x can be transformed by *either* F or G.

by simply ensuring that the tail nodes of arcs representing the subsequent operations are properly placed.

It is important to point out the difference between Figures 4-6(a) and 4-6(b). In Figure 4-6(a), the data block x is transformed by the single-input dual-output primitive C, generating both data blocks y and z. In Figure 4-6(b), on the other hand, the data block x can be transformed by *either* F or G, but not both; as result, only one of the data blocks y and z is generated.

4.3.2 Interdependent Decision Points

When an alternative construct operates on multiple input blocks, its associated decision nodes will not be independent. For example, consider the algebraic expression

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = (\begin{bmatrix} A \\ C \end{bmatrix} + \begin{bmatrix} B \\ D \end{bmatrix}). * \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{4.3}$$

The nodes representing x_1 and x_2 are both decision points, but their decisions must be coordinated. Graphically, a group of interdependent decision points is represented via a dotted line surrounding all of the nodes in the group, as shown in Figure 4-7(a). Figure 4-7(b) depicts a structurally similar algorithm graph without the dependency grouping; its corresponding algebraic expression is

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} A + B \\ C + D \end{bmatrix} . * \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{4.4}$$

Any graph with interdependent decision nodes can be graphically transformed to one without the interdependence by inserting extra arcs representing two special operations, Combine and Split. Combine groups its input nodes into one, while Split performs the inverse operation. Algebraically, we can use these operators to eliminate alternative expressions operating on vectors of data blocks.

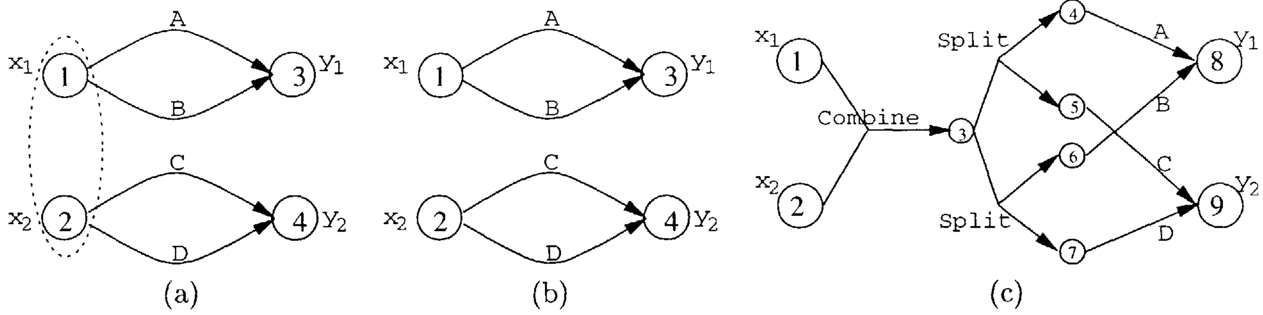


Figure 4-7: (a) Algorithm graph for the expression of (4.3) (b)Algorithm graph for expression of (4.4) (c) Transformation of graph (a) into one without interdependent nodes.

For example, the expression of (4.3) can be modified thus:

$$\begin{aligned}\begin{bmatrix} \mathtt{y_1} \\ \mathtt{y_2} \end{bmatrix} &= (\begin{bmatrix} \mathtt{A} \\ \mathtt{C} \end{bmatrix} + \begin{bmatrix} \mathtt{B} \\ \mathtt{D} \end{bmatrix}).* \begin{bmatrix} \mathtt{x_1} \\ \mathtt{x_2} \end{bmatrix} = (\begin{bmatrix} \mathtt{A} \\ \mathtt{C} \end{bmatrix} + \begin{bmatrix} \mathtt{B} \\ \mathtt{D} \end{bmatrix})\mathtt{SplitCombine} \begin{bmatrix} \mathtt{x_1} \\ \mathtt{x_2} \end{bmatrix} \\ &= (\begin{bmatrix} \mathtt{A} \\ \mathtt{C} \end{bmatrix} \mathtt{Split} + \begin{bmatrix} \mathtt{B} \\ \mathtt{D} \end{bmatrix} \mathtt{Split})\mathtt{Combine} \begin{bmatrix} \mathtt{x_1} \\ \mathtt{x_2} \end{bmatrix}\end{aligned}$$

The last equality is due to the Distributive Law 1 in Section 4.2.1. The result of these manipulations is that the alternative construct within the parenthesis now operates on the single output of the `Combine` operation, and hence the interdependence is eliminated. Because the `Combine` and `Split` operations are inverses of one another, the transformation preserves the expression of the algorithmic progression.

This process can be performed directly on the graph as well. For concreteness of discussion, suppose that the graph has n interdependent decision nodes, each with m alternatives. First, create an n-input 1-output `Combine` arc whose inputs are the interdependent decision nodes. Then, create m 1-input n-output `Split` arcs, and attach their input nodes to the combined node. Now each of the m alternatives can be attached to the output set of a single `Split` arc. Figure 4-7(c) depicts the transformed graph of Figure 4-7(a).

4.3.3 Examples

As a simple example, Figure 4-8 shows the algorithm graph corresponding to the expression

$$\mathtt{y} = ((\mathtt{DB} + \mathtt{EC})\mathtt{A} + \mathtt{I} \begin{bmatrix} \mathtt{G} \\ \mathtt{H} \end{bmatrix} \mathtt{F})\mathtt{x}$$

In the graph, node 1 corresponds to the input data block `x`, while node 10 corresponds to the output `y`. The graph shows that primitive `A` is a single-input, dual-output operator, while primitive `B` is a dual-input, single-output operator. Additionally, there are three functionally equivalent algorithms

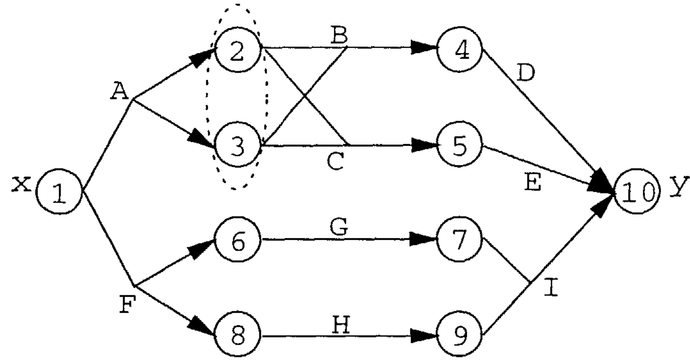


Figure 4-8: Algorithm graph corresponding to expression $y = ((\mathtt{DB} + \mathtt{EC})\mathtt{A} + \mathtt{I}\begin{bmatrix}\mathtt{G}\\ \mathtt{H}\end{bmatrix}\mathtt{F})\mathbf{x}$.

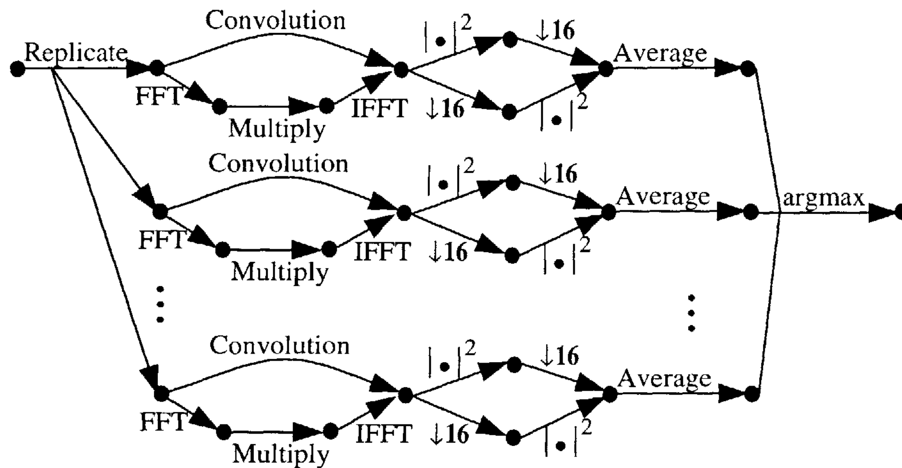


Figure 4-9: Algorithm graph of a modulated filter bank implementation of periodogram averaging to find the peak frequency of a spectrum.

for processing x. The first corresponds to the top-most path requiring the sequential application of operators A-B-D, while the second is the path A-C-E. The third algorithm requires F, then G and H concurrently, and finally I. The first decision point is at node 1, where the system chooses between A and D. A second decision point involves the interdependent nodes 2 and 3, which must both choose B or both choose C.

To show the construction of an algorithm graph for a signal processing task, Figure 4-9 shows the algorithm graph for a modulated filter bank implementation of periodogram averaging. The block diagram of the algorithm is in Figure 4-1(a), and its corresponding algebraic expression is the first line of Figure 4-4. Because of the large number of nodes in this graph, the node numberings are not shown in this example.

In the algorithm graph of Figure 4-9, the signal, represented by the leftmost node, is first repli-

cated. Each of the replications is filtered independently via one of two methods: either through direct convolution, or through multiplication in the frequency domain. This is graphically represented by the fact that the "Convolution" arc connects the same two nodes as the "FFT-multiply-IFFT" path. The next two operations on the filter outputs are downsampling and finding the magnitudes of the samples. Since these two operations are commutative, there are parallel paths showing the two orders of execution. Finally, the channel with highest average output magnitude is found as the peak of the spectrum.

4.3.4 Terminology, Assumptions, and Notation

Within an algorithm graph, nodes with no incoming arcs are the inputs of the algorithms, and those with no outgoing arcs are the outputs. An algorithm graph composed of nodes and arcs as we describe is therefore hierarchical in the sense that each arc can be further represented by another algorithm graph. For simplicity, however, all algorithms presented to the system are assumed to be SISO, although the arcs within the graph may be of any form. This is not a limiting assumption, since all of the input data blocks of a multi-input function can be grouped into one, and then become separated into the desired forms at the beginning of the algorithm. Similarly, the outputs of a multiple-output function can be followed by a grouping function at the very end to form a single-output function. For convenience, the input and output nodes of the algorithm graph are also called its origin and destination nodes, respectively. Nodes other than the origin and destination nodes are called internal nodes.

The graphical construct offers a simple visualization of the state of a data block as it gets processed. The node(s) in the graph corresponding to the current state(s) is(are) called *active*. When the data block is introduced into the network, only its input node is active. The number of active nodes increases following SIMO functions and decreases following MISO functions. The system cannot proceed with a multi-input operation until all input nodes of the operator are active. Active nodes with more than one outgoing arc calls for the system to choose among the processing alternatives. For example, in Figure 4-8, at the beginning of processing, node 1 is active. If the bottom path is chosen, then nodes 6 and 8 are active following processing by function `F`. Following `G`, node 7 is active; following `H`, node 9 is active. When both nodes 7 and 9 are active, an application of `I` leads us to the completion of processing at node 10.

Graphs without interdependent decision nodes are often simpler to analyze. Hence, the availability of the transformation of Section 4.3.2 simplifies the analysis of many algorithm graphs. Therefore, unless otherwise stated, we limit our discussion to those graphs *without* interdependent decision nodes.

Mathematically, an algorithm graph $\mathcal{G}$ can be represented by its set of nodes $\mathcal{N} = \{1, 2, ..., N\}$ and arcs $\mathcal{A}$. The origin and destination nodes are always numbered 1 and N, respectively.

Each arc $a \in \mathcal{A}$ can be written as (T_a, H_a, p_a), where T_a represents the ordered set of tail nodes of the arc, H_a represents the ordered set of head nodes of the arc, and p_a is the processor type associated with the arc. Let $|T_a|$ and $|H_a|$ denote the cardinality of the sets T_a and H_a, respectively. For each node n, let $I_n = \{a \in \mathcal{A} | n \in H_a\}$ and $O_n = \{a \in \mathcal{A} | n \in T_a\}$. I_n and O_n are the sets of incoming and outgoing arcs at node n, respectively.

4.3.5 Decomposition

An algorithm graph $\mathcal{G}$, like its associated algebraic expression, represents a set G of functionally equivalent algorithms which transform its input to its output. Each member of G is in turn represented algebraically by an expression with only sequential and concurrent constructs, and graphically by a subgraph corresponding to that expression. In this section, we discuss the decomposition of algorithm graphs and expressions into their component execution paths.

Simple Graphs

We begin by considering a class of algorithm graphs with trivial decompositions. An algorithm graph is *simple* if it satisfies the following characteristics:

- Every node except the destination node has exactly one outgoing arc.
- Every node except the origin node has exactly one incoming arc.

An example is shown in Figure 4-10.

A simple graph represents a single implementation for achieving a processing goal. It can not be further decomposed because it contains no decision points. In fact, it can be readily translated into an algebraic expression with only sequential and concurrency constructs via a procedure of back-substitution. First, associate each node with a data block variable. For each arc in the graph, we can write an "equation" of the forms in Figure 4-2, relating the input and output nodes of each arc. The two characteristics above guarantee that every internal node variable appears exactly twice in the resulting system of equations–once on the left side of an equation, and once on the right side. As a result, starting with the equation involving the destination node, a process of substitution can obtain a sequence of expressions which relate the destination node to the successfully "earlier" intermediate nodes of the graph, until the origin node is eventually reached. The only exception to this process is the possible need to insert `Permutation` and `Identity` operators to ensure the correct arrangement of the vector of operands. At the end of the process, Distributive Law 4 described in Section 4.2.1 may be applied to simplify the expression.

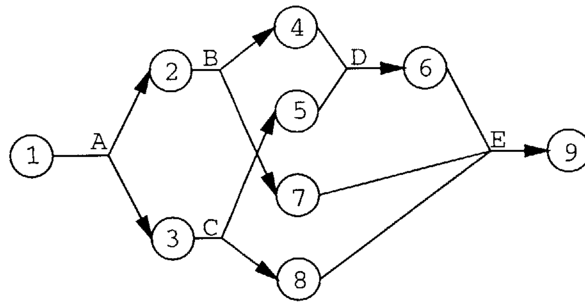


Figure 4-10: A *simple* algorithm graph.

For example, the algorithmic expression for the graph in Figure 4-10 can be obtained via the following process:

$$\begin{aligned}
\mathtt{x_9} &= \mathtt{E}\begin{bmatrix} \mathtt{x_6} \\ \mathtt{x_7} \\ \mathtt{x_8} \end{bmatrix} \\
&= \left(\mathtt{E}\begin{bmatrix} \mathtt{D} & \\ & \mathtt{Identity} \\ & & \mathtt{Identity} \end{bmatrix}\right) .* \begin{bmatrix} \begin{bmatrix} \mathtt{x_4} \\ \mathtt{x_5} \end{bmatrix} \\ \mathtt{x_7} \\ \mathtt{x_8} \end{bmatrix} \\
&= \left(\mathtt{E}\begin{bmatrix} \mathtt{D} & \\ & \mathtt{Identity} \\ & & \mathtt{Identity} \end{bmatrix} .* \mathtt{Permutation\text{-}1324}\right)\begin{bmatrix} \mathtt{x_4} \\ \mathtt{x_7} \\ \mathtt{x_5} \\ \mathtt{x_8} \end{bmatrix} \\
&= \left(\mathtt{E}\begin{bmatrix} \mathtt{D} & \\ & \mathtt{Identity} \\ & & \mathtt{Identity} \end{bmatrix} .* \mathtt{Permutation\text{-}1324}\begin{bmatrix} \mathtt{B} & \\ & \mathtt{C} \end{bmatrix}\right) .* \begin{bmatrix} \mathtt{x_2} \\ \mathtt{x_3} \end{bmatrix} \\
&= \left(\mathtt{E}\begin{bmatrix} \mathtt{D} & \\ & \mathtt{Identity} \\ & & \mathtt{Identity} \end{bmatrix} .* \mathtt{Permutation\text{-}1324}\begin{bmatrix} \mathtt{B} & \\ & \mathtt{C} \end{bmatrix} .* \mathtt{A}\right) \mathtt{x_1}
\end{aligned}$$

Decomposition into Execution Paths

Generally, however, an algorithm expression and its graphical representation $\mathcal{G}$ represents a set of algorithms.

Using Distributive Laws 1, 2, and 3 from Section 4.2.1, one can expand any algorithm expression into the form

$$\mathrm{y} = (G_1 + G_2 + ...G_n)\mathrm{x}, \tag{4.5}$$

where each G_i is an algebraic expression with only sequential and concurrent constructs, and the set $G = \{G_i\}$ is the functionally equivalent set of algorithms represented by the original expression. An expression of a set of functionally equivalent algorithms in the form of (4.5) is called an *expanded form.*

Graphically, every G_i is represented by a subgraph $\mathcal{G}_i$ satisfying the following conditions:

- $\mathcal{G}_i$ is a connected subgraph of $\mathcal{G}$; every node and arc in $\mathcal{G}_i$ is also in $\mathcal{G}$.
- $\mathcal{G}_i$ has the same origin and destination nodes as $\mathcal{A}$, and no others.
- $\mathcal{G}_i$ is simple.

A subgraph satisfying these conditions is called an *execution path* of $\mathcal{G}$. (Notice that an execution path typically does not fit the conventional graph theory definition of a "path".) The algebraic expression of $\mathcal{G}_i$ can be obtained independently from G_i via the process of back-substitution described earlier. There is a one-to-one correspondence between the set of execution paths of $\mathcal{G}$ and the members of the expanded form of the algebraic expression.

In the context of algorithm adaptation, execution paths are the basic decision variables of the system. For a given stream of data blocks tagged with the same algebraic expression, the system must allocate the data blocks among the set of execution paths.

4.3.6 Considerations of Validity

While the algorithm graph is a rich representation of algorithms, not all interconnections of nodes and arcs form valid, computable algorithm graphs. In the context of the thesis, the interconnections in an algorithm graph are valid if

1. All possible sequences of processing steps denoted by these connections successfully transform the origination node to the destination node;
2. In any processing sequence, no node is active more than once.

If condition 1 is violated, there exists a processing sequence which *deadlocks* and cannot proceed onto the destination node. If condition 2 is violated, there exists a data state (represented by a node) which the system reaches more than once, wasting system resources. Invalid algorithm graphs unnecessarily burden the system by causing it to work on jobs whose computation is redundant or not completeable.

The validity conditions can be verified through the the successful completion of all possible progressions of the following algorithm:

1. $A := \{1\}$

2. If $H_a \cap A = \emptyset$ for each $n \in A$ and $a \in O_n$, proceed to Step 3. Otherwise the algorithm graph is *invalid.*

3. Find an arc a such that $T_a \subset A$. If no such arc exists, the algorithm graph is *invalid.* Otherwise, $A := A/T_a \cup H_a$. Mark a and H_a. If $A = \{N\}$, stop; otherwise repeat Step 2.

The algorithm emulates the progression of the state of the data as the system processes it. In each iteration, the set A can be regarded as the set of active nodes during processing. Step 1 initializes the process by setting the origination node active. In Step 3, the system chooses an arc for which all input nodes are active and ready for the next processing step. It marks this arc and its head nodes, and updates A to reflect the new set of active nodes. Depending on the sequence of arcs marked in each iteration, all possible execution paths can be generated. However, the graph is considered invalid if even one progression of the algorithm leads to that determination.

The algorithm graph may be determined to be invalid in either Step 2 or 3. In Step 2, invalidity is established if there exists an outgoing arc of an active node for which at least one head node is already active. A further progression on this particular arc would lead to the violation of Condition 2 above. An example of such a graph is Figure 4-11(a). In this graph, after marking arc `A`, nodes 2 and 3 are both active. Then, node 4 can become active twice, once via arc `B` and once via arc `C`. Since this forces the system work to twice as hard as actually necessary, we invalidate graphs which leads to such situations.

Step 3 fails if there is no arc for which all input nodes are active. In other words, there exists a data state progression which stalls and cannot reach the destination node. Such a graph is clearly invalid, and an example is shown in Figure 4-11(b). In this graph, either node 2 or 3, but not both, can be active at the same time. As a result, arc `F` can never be activated, and the processing in effect stalls.

Upon successful completion of this algorithm, the marked nodes and arcs represent an execution path of the graph. Step 2 guarantees that every marked node has exactly one marked incoming arc, and Step 3 guarantees that every marked node has only one marked outgoing arc. Since the algorithm begins with origination node and concludes on the destination node, the result satisfies

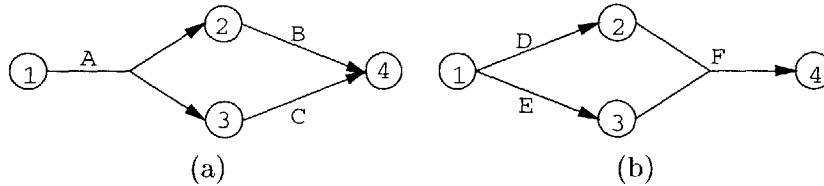


Figure 4-11: Two invalid graphs.

the definitions of an execution path. Hence, in a valid graph where every possible progression of this algorithm completes successfully, the system can manipulate the data from the origination to the destination state without wasting system resources, regardless of the chosen processing sequence. In this thesis, we consider only those algorithm graphs satisfy this property.

4.4 Algorithm Graphs Compared to Communication Networks

Thus far in this chapter, the constructs of the algorithm description language and the algorithm graph have been presented only in the syntactical sense. However, beyond their importance for defining the interface between the users and the system, algorithm graphs also motivate an interesting comparison between computation and communications. The structure of algorithm graphs is reminiscent to that of data communications networks; whereas data packets are transmitted between nodes in a communications network, they are processed between nodes in the algorithm graphs. Whereas routing in communications means choosing one set of links over others for transmission, in the algorithm context it means choosing one particular implementation over others. This similarity leads us to use the terms algorithm "network" and "graph" interchangeably.

One of the central concepts in the design of data networks is that of layering, by which the functions of the network are modularized and divided hierarchically into multiple layers [6]. Under the concept, the proper specification of the service provided by each layer and the interface between layers allows each layer to be designed independently. Each layer simply utilizes the services of the next lower level to implement its pre-specified functions, which in turn are used by the next higher level of the hierarchy.

In this section, we examine the similarities and differences between communications networks and algorithm graphs. For clarity, the discussion is structured according to the standard OSI layering model of communications networks[6][1]. The goals of this comparison between algorithm and communications networks are to enhance our understanding of algorithm graphs as well as to suggest methodology for formulating and solving problems related to them. Indeed, we find that

[1]The OSI model is used for convenience and clarity of discussion. It should be noted that this model is only followed loosely in the designs of actual communication networks today.

this interpretation of algorithms graphs can be extended to a mathematical formulation with which the system can dynamically choose among a set of functionally equivalent algorithms to minimize system congestion.

4.4.1 Physical Layer

The lowest level of a communications network is the physical layer, which performs the most basic function of the network: the actual transmission of data from link to link across some physical medium. It is often physically implemented in a modem, and provides an unreliable bit pipe between two connected nodes in the network. This bit pipe can be characterized by its speed and error rate. Similarly, in the context of an algorithm network, the physical layer refers to the actual processing of a data block by a physical processor, which is also characterized by its processing rate and reliability.

There are two fundamental differences between the physical layers of the two different types of networks, however. Four different types of arcs are used in algorithm graphs, denoting different types of processors which may be multi-input or multi-output. It is clear that SISO arcs of the algorithm graph are analogous to point-to-point communications channels, and SIMO arcs are analogous to broadcast channels, such as those used for radio or television, in which data is sent from a single transmitter to multiple receivers. On the other hand, there may not be natural equivalences to the MISO and MIMO arcs of algorithm graphs in the communications context.

A second difference lies in the fact that the size of a data packet cannot be changed during the process of transmission through a communications channel, while processing may very well change the size of data blocks. Some simple examples of such processing are sampling-rate conversion and data compression. This introduces an additional layer of complexity when the system chooses among the set of equivalent algorithms, since the data block size must be tracked as processing proceeds, in order to accurately predict how much work or time will be needed to complete each step of processing.

4.4.2 Data Link Control Layer

The data link control (DLC) layer of a data network contains error correction protocols which converts the unreliable bit pipe provided by the physical layer into an error-free one. Following the transmission of a packet in the physical layer, the DLC layer detects errors in the packet via checks such as parity check and cyclic redundancy check codes. If errors are detected, the protocol may either request retransmission or drop the packet and leave the retransmission to be attempted on an end-to-end basis in a higher network layer.

In the context of an algorithm network, we can conceptualize a similar DLC layer which contains a set of fault-tolerant computing strategies. Following the processing of a data block by a physical

processor, the protocols in this layer checks for errors via schemes ranging from replicate-and-vote protocols to other sophisticated methods similar to those of error correction coding in communications [2]. If errors are detected, the system may choose to re-process the data block with the same primitive operator; such a protocol would require that a copy of the data block before processing is kept at each step until the processing output has been verified. On the other hand, we may choose a protocol in which such data blocks are dropped, and re-processing is attempted only from the beginning of the entire algorithm.

While retransmission in the communications context is generally construed to be over the *same* channel, however, in the computation context, the system has the option of re-processing the data at a different physical processor. In fact, this option may be more viable, since hardware failures are more likely to be due to permanent faults, and transmission errors are more likely to be due to transient conditions in the channel.

4.4.3 Network Layer

The data link layer provides an error-free bit pipe between different nodes in the network. On top of it is the network layer, which is responsible for routing and flow control in the communications context. Routing is the task of selecting a sequence of links for transporting data packets from any origination node to any destination node. Flow control is the task of preventing over-congestion in the network, and is shared with the transport layer. All network layer processes associated with each node in the network communicate and cooperate in order to accomplish these functions. As a result, it is perhaps the most complex layer in terms of algorithmic and control issues. The network layer provides an virtual bit pipe between any two origin and destination nodes in the network, allowing peer processes for higher layers to communicate directly from end-to-end.

Given the large number of links and paths, the task of routing packets from their origin to their destination is one of the most complex and significant functions of a communications network. The task requires collaboration among all of the nodes of the network, and it is run continuously during operation, to handle new sessions as well as to modify routes in response to link and node failures and congestion. At the same time, a good routing algorithm is central to achieving high performance in the network. It increases overall throughput, decreases the delay experienced by packets, and makes the network robust to unreliable links and nodes.

In the processing context, the input and output states of a data block correspond to the origin and destination nodes of its associated algorithm graph, respectively. When the system chooses a single algorithm out of the full functionally equivalent set represented by the algorithm graph, it performs a function analogous to routing; it is in effect choosing a simple subgraph on which the data block "travels" on. In this thesis, this task of execution path selection is also called *algorithm adaptation*, to be fully described in Chapter 5. Routing and algorithm adaptation are

both resource allocation problems. Each arc of the algorithm graph is analogous to a link in an algorithm "network", and data blocks are analogous to data packets which need to be routed from an origin to a destination. With multiple data streams and multiple algorithms, the analogy can be extended to routing through a highly complex network with many different origin-destination pairs. Just as a good routing algorithm is central to achieving high performance on the network, a good algorithm adaptation scheme results in an efficient and reliable computing system.

In networking literature, there exist numerous routing strategies, classifiable through various criteria, and each with a parallel concept for algorithm adaptation. One fundamental classification separates routing algorithms to be either *static* or *adaptive*. Static strategies fix the path used for each origin-destination pair regardless of the traffic patterns, and are used only in very simple networks or in networks where efficiency is not crucial. Adaptive strategies adapt the path assignments in response to congestion, and are used in most modern networks. In the processing context, a static strategy corresponds to what is used on most computing systems today; for each algorithm graph, the user chooses one simple subgraph ahead of time, regardless of the traffic patterns. An adaptive strategy changes the algorithm chosen for the packets according to the system state.

The adaptive routing strategies can be further classified according to the frequency with which path decisions are made for each origin-destination pair of the network. In *virtual circuit* routing, a routing decision is chosen during initialization of the virtual circuit, and all subsequent packets of the virtual circuit use the same path. On the other hand, in *datagram* routing, a decision is made for each packet individually. Algorithmically speaking, applying virtual circuit routing on an algorithm graph involves a single decision for each data stream; a simple subgraph for processing all of the data packets of a data stream is chosen when the data stream is initialized. A datagram routing strategy would decide on the algorithm for each data packet individually. The complex dynamics of network flow makes datagram routing significantly more difficult than virtual circuit routing. While most networks today use virtual circuit routing, datagram routing is used on the Internet.

It is also interesting and instructive to consider analogies which extend in the "opposite" direction. Consider a computing system in which all algorithms are simply fully coded, rather than expressed as a composition of smaller primitive building blocks. Graphically, then, each algorithm graph consists of a single arc connecting the input and output nodes. A communications network equivalent to this type of system is one in which every origin-destination pair is connected by a single direct link, and no routing is used. Such networks are rare because they fail to offer the benefits of resource sharing. A failed link stops all communications between the affected pair; an idle link can not be used for other purposes. And yet, today's computing systems are largely built this way, with every job completeable in only one way.

There are two key differences between the routing tasks in communications versus algorithms,

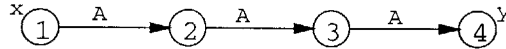


Figure 4-12: Algorithm graph for y = AAAx.

however. First, consider the very simple algorithm graph shown in Figure 4-12. In the graph, the data block is processed by primitive operator A three different times in sequence. Even though the data block traverses through three *different* arcs, it uses the *same* resource three times. This situation rarely occurs in a communications network, where distinct arcs generally refer to distinct resources.

The second difference stems from the fact that each arc in an algorithm graph refers to a *type* of primitive processor rather than a specific physical processor. In fact the task of choosing a physical processor can be considered to be yet another routing problem which involves looking at the structure of the underlying communications network. In this thesis, we assume that the underlying communications network can handle requests to transmit a data block to any physical processor.

4.4.4 Transport Layer

The network layer in effect provides a bit pipe between origin-destination pairs in the network. The transport layer handles a variety of issues, the most important of which is flow control. The task of flow control, shared between the network and transport layers, is that of restricting the externally offered load to a level which is manageable by the network. Such techniques often entail the blocking or discarding of packets or session when it is determined that the network is overly congested.

In the context of computation, "flow control" can be accomplished in a variety of ways. Certainly, it is possible for the system to block or discard packets when it is overly congested. Less drastic measures are also available, however. The class of approximate processing techniques may be applied to gradually relax the processing objectives and simplify or shorten algorithms in response to system overload [27]. A closely related concept in communications is the technique of embedded coding which allows the network to gracefully reduce the fidelity of data packets on the network in response to congestion.

4.5 Summary

The goal of this chapter is to introduce a way of expressing and thinking about algorithms. The algorithm description language is designed to allow the efficient expression of multiple equivalent

implementation choices in terms of the primitive operations available in the processing network. It is an intuitive interface to the system, and permits the processing steps to be specified to the system at a level higher than machine-level code, leaving implementation details to individual hardware platforms. It is also naturally visualized using an algorithm graph in which the nodes represent data states and the arcs represent transformations of the data.

Unlike existing forms of data flow graphs, algorithm graphs are capable of expressing concurrency and equivalence simultaneously with no ambiguity. Every algorithm graph can be decomposed into a set of simple subgraphs, or execution paths, each representing one of the functionally equivalent algorithms described by the graph. In an valid algorithm graph, all possible sequences of processing steps denoted by the interconnection of nodes and arcs successfully transform the data from its input state to its desired output state. Moreover, no data state is obtained or reached more than once within a processing sequence.

Beyond their syntactical significance, algorithm graphs also point to an interpretation of computation as analogous to communication networks. The transmission of bits across a physical link is analogous to the transformation of data by a processor. The error detection and correction protocols in a communications network are similar to fault-tolerant computing schemes. Packet routing is analogous to execution path selection in algorithm adaptation. This comparison between communications and algorithm networks stimulates much of our work in this thesis, especially in the formulations of algorithm adaptation and system capacity.

Chapter 5

Algorithm Adaptation

Thus far in the thesis we have developed a model of the computing environment and a representation of signal processing tasks in terms of algorithm graphs. The system model, developed in Chapter 3, captures the characteristics of a dynamic, heterogeneous DSP system as a collection of specialized virtual processors. The task representation, developed in Chapter 4, organizes collections of functionally equivalent algorithms as algorithm "networks" on which data packets are transformed as they move across operator arcs. This chapter builds on this framework to develop the technique of algorithm adaptation, which allows the system to choose among a set of functionally equivalent algorithms to apply to the given data set.

Section 5.1 motivates algorithm adaptation strategy by describing the potential benefits on overall system performance. Section 5.2 describes the first of two strategies for algorithm adaptation, minimum cost packet-wise adaptation. In this strategy, each virtual processor in the system is assigned a cost according to its processing capacity and current load. Then, based on these cost measures, the system chooses the execution path incurring the least cost for each data block individually. The second strategy, optimal system-wide algorithm adaptation, is described in Section 5.3. This strategy formulates the algorithm adaptation problem into a constrained nonlinear optimization problem in which a measure of system congestion is minimized. The numerical solutions for the optimization problem can be obtained iteratively using a gradient projection algorithm.

5.1 Motivations

Algorithm adaptation is a technique for adapting algorithms to the computation environment. The user presents the system with sets of functionally equivalent algorithms, and leaves the system to choose among them. Instead of relying on the user to choose efficient algorithms which would not place undue burden on the system, the system itself chooses the algorithm dynamically, according to its own optimization criterion. As a result, it can be viewed both as a resource allocation technique

from the system perspective, and as a dynamic algorithm optimization technique from the user perspective. It takes advantage of the additional degree of freedom afforded by considering sets of functionally equivalent algorithms to alleviate the user's inability to fully optimize his algorithms due to the time-variance and heterogeneity of the computation environment.

The development of algorithm adaptation is based on the premise that the task of algorithm optimization is better accomplished by the system than by the user. In dynamic, heterogeneous computing environments, this premise holds for a few reasons. First, because the user lacks specific, detailed knowledge of the system state, it is only possible to guess its load, its bottlenecks, and an appropriate optimization criterion. For a job with a long lifetime, the system state may change during its execution, making the user's optimizations outdated. The system, on the other hand, is capable of monitoring and adjusting its own state continuously. Once given the options for executing its tasks, it becomes capable of choosing algorithms based on its current state.

Beyond superior knowledge of its own state, the system also has optimization options which would not be available to the user. For example, an individual user who has no control over the processing demands of other users can only optimize with respect to his own job, sometimes at the cost of others. When the system performs algorithm adaptation, however, it can use a system-wide optimization criterion which simultaneously minimizes the execution times of multiple tasks, improving the overall efficiency of the system.

A third benefit of delegating algorithm optimization to the system is convenience to the user. His only task is to specify sets of functionally equivalent algorithms, which may be facilitated by software systems. The system configuration becomes transparent to him, because the same code can be used whether the system is just a single processor or a super computer. He is relieved of the particularly difficult task of predicting the state of a multi-user, multi-processor system.

Finally, the ultimate benefits of algorithm adaptation are exactly those of adaptive routing in communications: efficiency and robustness. The system can operate more efficiently because system bottlenecks can be avoided. Robustness is achieved by reducing dependence of any task on the availability of specific processors. The system is given more flexibility with which it can handle a large variety of demands.

Given these advantages, how can algorithm adaptation be implemented? Like any other adaptation problem, there are no doubt numerous answers to this question. In this thesis, our explorations begin with the observations of the structural similarities between data networks and algorithm graphs in Section 4.4. As discussed there, despite significant differences, algorithm adaptation on an algorithm graph is conceptually analogous to routing on a data network. Hence, one approach is to use existing routing algorithms as a springboard. In this chapter, we discuss algorithm adaptation approaches derived from two classes of routing algorithms, shortest path routing and optimal routing.

5.2 Minimum Cost Packet-wise Adaptation

Shortest path, or minimum cost, routing is perhaps the most commonly used routing strategy in communications networks today. Its operation involves assigning a cost ("length") to every link in the network, and sending every packet along the minimum cost ("shortest") path from the origin and the destination nodes of the packet. The cost associated with each link may be derived from its speed, its load, or some other factor. As a simple example, in some networks, every link is assigned unit length, so that the resulting algorithm sends every packet on the minimum-hop path involving the least number of links. In more sophisticated networks, the link lengths change over time in response to congestion, failure, or other events, giving rise to *adaptive shortest path routing.* To calculate the shortest path, each node in the network keeps a routing table, which keeps track of the shortest distance between it and any other destination node. Periodically, a node will routing updates to their neighbors, which in turn use the new information to update their respective routing tables.

Conceptually, the shortest path criterion is easily extended to algorithm adaptation. In such a system, every virtual processor is assigned a cost, and the algorithm adaptation process simply chooses the minimum cost execution path for processing each data block. The cost for using a virtual processor is chosen to reflect its state. Periodically, the processor costs are broadcast throughout the network, and subsequent execution path selections reflect the new system state.

A variety of algorithms exist for solving for the shortest path between two points on a graph. In the communications context, some of the more famous algorithms for shortest path routing are the Bellman-Ford, Dijkstra, and Floyd-Marshall algorithms [6]. Most of these algorithms apply the dynamic programming principle:

$$\begin{pmatrix}\text{cost at}\\ \text{node } k\end{pmatrix} = \min_{i:\ (i,k)\text{ is an arc}} \left\{ \begin{pmatrix}\text{cost at}\\ \text{node } i\end{pmatrix} + \begin{pmatrix}\text{cost of}\\ \text{arc } (i,k)\end{pmatrix} \right\}, \tag{5.1}$$

where "cost at node i" refers to the minimum cost from the origin to node i. In directed acyclic graphs, the direct application of (5.1) to each node in topological order allows the minimum costs to be propagated from the origin to the destination node in $O(m)$ time, where m is the number of arcs in the graph.

In algorithm adaptation, the dynamic programming principle states that at each decision point of the algorithm graph, the branch incurring the least total cost is taken. In practice, however, a few complications arise. First, due to the existence of multi-input or multi-output arcs, the concept of the minimum cost from the origin to a given node k is not as easily defined. Consider a 2-input, 3-output arc. Even though the cost of traversing the arc is well-defined, the propagation of the cost of the 2 input nodes to its 3 output nodes is not a trivial concern. We must also account for the existence of dependent decision nodes in an algorithm graph. An algorithm which does not

perform this properly may derive shortest paths which are not valid execution paths.

Fortunately, the complex structure of algorithm graphs is also describable by its algebraic expression. Since the algebraic expression and the algorithm graph are equivalent expressions of sets of execution paths, the minimum cost execution path can be found on whichever structure is more convenient. In the algebraic expression, each "+" represents a decision point, with its operands being the alternatives. By choosing the minimum cost alternative at each decision point, the overall cost of the algebraic expression is minimized. This process can be formalized by a simple set of recursive rules.

5.2.1 Notation

To facilitate the description of these rules, we introduce some additional notation.

For any n-input m-output primitive operator P, suppose that its input and output nodes are ordered, and a reference node is chosen among the input nodes. We define the following:

- $k_{\mathtt{P}}$: the node number of the reference input among the inputs.
- $S_{\mathtt{P}}$: the input data size transition vector of P with respect to the reference input. That is, if Ψ_{in} is the vector of input data sizes, and $\Psi_{k_{\mathtt{P}}}$ is the data size at the reference input, then

$$\Psi_{\text{in}} = S_{\mathtt{P}} \Psi_{k_{\mathtt{P}}}.$$

- $T_{\mathtt{P}}$: the output data size transition vector of P with respect to the reference input. That is, if Ψ_{out} is the vector of output data sizes, and $\Psi_{k_{\mathtt{P}}}$ is the data size at the reference input, then

$$\Psi_{\text{out}} = T_{\mathtt{P}} \Psi_{k_{\mathtt{P}}}.$$

- $C_{\mathtt{P}}$: the cost per data size unit of applying P. In this thesis, we make the simplifying assumption that the cost of processing a data block scales linearly with the size of the inputs. Therefore, if Ψ is a vector describing the sizes of the input data blocks of P, then

$$\text{cost of using } \mathtt{P} = C_{\mathtt{P}} \sum_{i=1}^{n} \Psi^{i},$$

where Ψ^{i} refers to i-th element of Ψ.

Finally, we define the following three functions:

- mincost($\mathtt{A}, \Psi$): minimum cost of executing the algebraic expression A on an input data set of size Ψ.

- $\text{minpath}(\mathtt{A}, \Psi)$: algebraic expression for the minimum cost path achieving $\text{mincost}(\mathtt{A}, \Psi)$. $\text{minpath}(\mathtt{A}, \Psi)$ contains no alternative constructs, and

$$\text{mincost}(\text{minpath}(\mathtt{A}, \Psi), \Psi) = \text{mincost}(\mathtt{A}, \Psi)$$

- $\text{size}(\mathtt{A}, \Psi)$: vector of data block sizes resulting from applying the algebraic expression $\mathtt{A}$ on an input data set of size $\Psi_{\mathtt{A}}$.

These functions are well-defined only if the length of Ψ equals the number of inputs of $\mathtt{A}$.

5.2.2 Rules

We now present four rules for computing the mincost, minpath, and size functions for primitive operators, sequential, concurrent, and alternative constructs. By combining these rules, the minimum cost path for any algorithmic expression can be found.

Rule 1 (Primitive Operators). *If* $\mathtt{P}$ *is a primitive operator,*

$$\text{mincost}(\mathtt{P}, \Psi) = C_{\mathtt{P}} \sum_{i=1}^{n} \Psi^i \quad (5.2)$$

$$\text{minpath}(\mathtt{P}, \Psi) = \mathtt{P} \quad (5.3)$$

$$\text{size}(\mathtt{P}, \Psi) = T_{\mathtt{P}} \Psi^{k_{\mathtt{P}}} \quad (5.4)$$

This is a simple, and mostly obvious, set of rules. (5.2) and (5.4) repeat the definitions of $C_{\mathtt{P}}$ and $M_{\mathtt{P}}$, while (5.3) states that the minimum cost path for a primitive operator is itself.

Rule 2 (Sequential Constructs).

$$\text{mincost}(\mathtt{A}_2\mathtt{A}_1, \Psi) = \text{mincost}(\mathtt{A}_2, \text{size}(\mathtt{A}_1, \Psi)) + \text{mincost}(\mathtt{A}_1, \Psi) \quad (5.5)$$

$$\text{minpath}(\mathtt{A}_2\mathtt{A}_1, \Psi) = \text{minpath}(\mathtt{A}_2, \text{size}(\mathtt{A}_1, \Psi))\text{minpath}(\mathtt{A}_1, \Psi) \quad (5.6)$$

$$\text{size}(\mathtt{A}_2\mathtt{A}_1, \Psi) = \text{size}(\mathtt{A}_2, \text{size}(\mathtt{A}_1, \Psi)) \quad (5.7)$$

Two operations in sequential constructs are both executed one after the other in two steps. For this reason, the overall cost is simply a sum of the cost incurred in each step, and the minimum cost path is a concatenation of the individual minimum cost paths. The data size vector is the same as that after the final step of the sequential construct.

Rule 3 (Concurrent Constructs).

$$\text{mincost}(\begin{bmatrix} \mathtt{A}_1 \\ \mathtt{A}_2 \end{bmatrix}, \begin{bmatrix} \Psi_1 \\ \Psi_2 \end{bmatrix}) = \text{mincost}(\mathtt{A}_1, \Psi_1) + \text{mincost}(\mathtt{A}_2, \Psi_2) \tag{5.8}$$

$$\textit{OR} \qquad \text{mincost}(\begin{bmatrix} \mathtt{A}_1 \\ \mathtt{A}_2 \end{bmatrix}, \begin{bmatrix} \Psi_1 \\ \Psi_2 \end{bmatrix}) = \max(\text{mincost}(\mathtt{A}_1, \Psi_1), \text{mincost}(\mathtt{A}_2, \Psi_2)) \tag{5.9}$$

$$\text{minpath}(\begin{bmatrix} \mathtt{A}_1 \\ \mathtt{A}_2 \end{bmatrix}, \begin{bmatrix} \Psi_1 \\ \Psi_2 \end{bmatrix}) = \begin{bmatrix} \text{minpath}(\mathtt{A}_1, \Psi_1) \\ \text{minpath}(\mathtt{A}_2, \Psi_2) \end{bmatrix} \tag{5.10}$$

$$\text{size}(\begin{bmatrix} \mathtt{A}_1 \\ \mathtt{A}_2 \end{bmatrix}, \begin{bmatrix} \Psi_1 \\ \Psi_2 \end{bmatrix}) = \begin{bmatrix} \text{size}(\mathtt{A}_1, \Psi_1) \\ \text{size}(\mathtt{A}_2, \Psi_2) \end{bmatrix} \tag{5.11}$$

Concurrent constructs are similar to sequential constructs in that every component of the construction must be executed, with the difference in the fact that the execution may take place in parallel. As a result, depending on the interpretation of the cost function, the method of accounting for the costs may vary. For example, if the cost function is system load, then all processing demands should be added up, so that (5.8) applies. On the other hand, suppose the cost function is processing time in a highly parallel environment. In this case, the time to execute all of the components of a concurrent construct is equal to the time taken by the slowest component. Within that time-frame, all of the other components would have also been computed, so that (5.9) applies. The choice between the two options depends on the system characteristics and design.

Rule 4 (Alternative Constructs).

$$\text{mincost}(\mathtt{A}_1 + \mathtt{A}_2, \Psi) = \min\{\text{mincost}(\mathtt{A}_1, \Psi), \text{mincost}(\mathtt{A}_2, \Psi)\} \tag{5.12}$$

$$\text{minpath}(\mathtt{A}_1 + \mathtt{A}_2, \Psi) = \arg \min_{\mathtt{B} \in \{\mathtt{A}_1, \mathtt{A}_2\}} \{\text{mincost}(\mathtt{B}, \Psi)\} \tag{5.13}$$

$$\text{size}(\mathtt{A}_1 + \mathtt{A}_2, \Psi) = \text{size}(\mathtt{A}_1, \Psi) \tag{5.14}$$

Finally, in alternative constructs, only one of the components need to be executed. As a result, the minimum cost execution path of the construct is the component incurring the lesser cost. The output data block size vector must be the same for all the alternatives, and hence can be computed via any component of the construct.

5.2.3 Example

As an example, consider a system described in Figure 5-1 and the algorithm expression

$$\mathtt{y} = ((\mathtt{DB} + \mathtt{EC})\mathtt{A} + \mathtt{I} \begin{bmatrix} \mathtt{G} \\ \mathtt{H} \end{bmatrix} \mathtt{F})\mathtt{x}. \tag{5.15}$$

Primitive P	C_P	M_P	Primitive P	C_P	M_P
A	1	$\begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}$	F	3	$\begin{bmatrix} 0.1 \\ 0.5 \end{bmatrix}$
B	2	$\begin{bmatrix} 0.5 & 0.5 \end{bmatrix}$	G	2	1
C	3	$\begin{bmatrix} 0.6 & 0.1 \end{bmatrix}$	H	0.6	1
D	2	2	I	1	$\begin{bmatrix} 5 & 1 \end{bmatrix}$
E	4	2			

Figure 5-1: Primitives and associated cost and data size transition matrix.

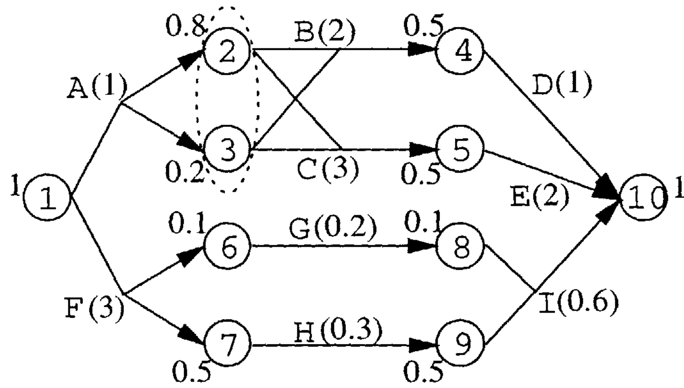


Figure 5-2: Algorithm graph for expression $\mathtt{y} = ((\mathtt{DB} + \mathtt{EC})\mathtt{A} + \mathtt{I}\begin{bmatrix} \mathtt{G} \\ \mathtt{H} \end{bmatrix}\mathtt{F})\mathtt{x}$.

To help with the visualization, Figure 5-2 shows the associated algorithm graph. The number next to each node is the data block size at that node, while the number in parenthesis next to each arc label corresponds the *total* cost incurred on that arc. For example, the data block sizes at nodes 2 and 3 are 0.8 and 0.2, respectively. Since $C_{\mathtt{B}}$ is 2, the cost for crossing B is

$$2(0.8 + 0.2) = 1$$

.

The minimum cost for executing the algorithm is

$$\text{mincost}((\mathtt{DB} + \mathtt{EC})\mathtt{A} + \mathtt{I}(\begin{bmatrix} \mathtt{G} \\ \mathtt{H} \end{bmatrix})\mathtt{F}, 1) = \min(\text{mincost}((\mathtt{DB} + \mathtt{EC})\mathtt{A}, 1), \text{mincost}(\mathtt{I}(\begin{bmatrix} \mathtt{G} \\ \mathtt{H} \end{bmatrix})\mathtt{F}, 1))$$

The first term of the minimization is

$$\begin{aligned}
\mathrm{mincost}((\mathtt{DB}+\mathtt{EC})\mathtt{A},1) &= \mathrm{mincost}(\mathtt{DB}+\mathtt{EC},\begin{bmatrix}0.8\\0.2\end{bmatrix})+\mathrm{mincost}(\mathtt{A},1)\\
&= \min(\mathrm{mincost}(\mathtt{DB},\begin{bmatrix}0.8\\0.2\end{bmatrix}),\mathrm{mincost}(\mathtt{EC},\begin{bmatrix}0.8\\0.2\end{bmatrix}))+\mathrm{mincost}(\mathtt{A},1)\\
&= \min(\mathrm{mincost}(\mathtt{D},0.5)+\mathrm{mincost}(\mathtt{B},\begin{bmatrix}0.8\\0.2\end{bmatrix}),\\
&\qquad \mathrm{mincost}(\mathtt{E},0.5)+\mathrm{mincost}(\mathtt{C},\begin{bmatrix}0.8\\0.2\end{bmatrix}))+\mathrm{mincost}(\mathtt{A},1)\\
&= \min(1+2,2+3)+1\\
&= 3+1\\
&= 4
\end{aligned} \tag{5.16}$$

The second term of minimization, if the concurrency cost is evaluated according to (5.8) is

$$\begin{aligned}
\mathrm{mincost}(\mathtt{I}\begin{bmatrix}\mathtt{G}\\\mathtt{H}\end{bmatrix}\mathtt{F},1) &= \mathrm{mincost}(\mathtt{F},1)+\mathrm{mincost}(\begin{bmatrix}\mathtt{G}\\\mathtt{H}\end{bmatrix},\begin{bmatrix}0.1\\0.5\end{bmatrix})+\mathrm{mincost}(\mathtt{I},\begin{bmatrix}0.1\\0.5\end{bmatrix})\\
&= 3+\mathrm{mincost}(\mathtt{G},0.1)+\mathrm{mincost}(\mathtt{H},0.5)+0.6\\
&= 3+0.2+0.3+0.6\\
&= 4.1
\end{aligned} \tag{5.17}$$

Since

$$\mathrm{mincost}((\mathtt{DB}+\mathtt{EC})\mathtt{A},1) < \mathrm{mincost}(\mathtt{I}\begin{bmatrix}\mathtt{G}\\\mathtt{H}\end{bmatrix}\mathtt{F},1)$$

We clearly have

$$\begin{aligned}
\mathrm{minpath}((\mathtt{DB}+\mathtt{EC})\mathtt{A}+\mathtt{I}\begin{bmatrix}\mathtt{G}\\\mathtt{H}\end{bmatrix}\mathtt{F},1) &= \mathrm{minpath}((\mathtt{DB}+\mathtt{EC})\mathtt{A},1)\\
&= \mathtt{DBA}
\end{aligned} \tag{5.18}$$

On the other hand, consider the results if the concurrency cost is evaluated according to (5.9).

(5.17) becomes

$$
\begin{aligned}
\texttt{mincost}(\texttt{I}(\begin{bmatrix} \texttt{G} \\ \texttt{H} \end{bmatrix})\texttt{F},1) &= \texttt{mincost}(\texttt{F},1) + \texttt{mincost}(\begin{bmatrix} \texttt{G} \\ \texttt{H} \end{bmatrix}, \begin{bmatrix} 0.1 \\ 0.5 \end{bmatrix}) + \texttt{mincost}(\texttt{I}, \begin{bmatrix} 0.1 \\ 0.5 \end{bmatrix}) \\
&= 3 + \max\{\texttt{mincost}(\texttt{G},0.1), \texttt{mincost}(\texttt{H},0.5)\} + 0.6 \\
&= 3 + \max\{0.2, 0.3\} + 0.6 \\
&= 3.9 \qquad (5.19)
\end{aligned}
$$

In this case,

$$\texttt{mincost}((\texttt{DB}+\texttt{EC})\texttt{A},1) > \texttt{mincost}(\texttt{I}\begin{bmatrix} \texttt{G} \\ \texttt{H} \end{bmatrix}\texttt{F},1),$$

and the minimum cost path becomes

$$\texttt{minpath}((\texttt{DB}+\texttt{EC})\texttt{A} + \texttt{I}\begin{bmatrix} \texttt{G} \\ \texttt{H} \end{bmatrix}\texttt{F},1) = \texttt{I}\begin{bmatrix} \texttt{G} \\ \texttt{H} \end{bmatrix}\texttt{F} \qquad (5.20)$$

In this example, the choice between (5.8) and (5.9) resulted in dramatically different execution path selections.

5.2.4 Discussion

Minimum cost adaptation shares the advantages and disadvantages of shortest path routing. Practically speaking, its implementation is simple, and as we will see in Chapter 7, it is capable of improving the overall throughput of the system. However, it is susceptible to the same drawbacks as adaptive shortest path routing: possible oscillatory behavior and limited throughput.

In minimum cost adaptation, after the processor loads are updated, all of the packets on a given data stream begin to use the most current minimum cost path, causing the processors required for that path to become heavily loaded, while leaving others idle. When the processor loads are updated again, the most current minimum cost path now switches to engage those processors which were previously idle, making those which were previously loaded idle. This process may then start to repeat itself. The execution path selections depend on the processor loads, which in turn depend on the execution path selections. Oscillations will occur if this feedback process is too "tight", but can be dampened by adding a variety of bias factors in the calculations.

A second disadvantage of minimum cost adaptation is that at any time, all of the blocks from the same data stream are processed using a single execution path. This can considerably reduce

the overall throughput of the system. For example, consider the very simple algebraic expression:

$$\mathtt{y} = (\mathtt{A} + \mathtt{B})\mathtt{x}$$

The maximum system throughput when processing data labelled with this algebraic expression is the sum of the processing capacities of primitives A and B, achieved when both execution paths are used to the full capacities. This is precluded in minimum cost adaptation, because only one of the execution paths can be chosen at any time.

More fundamentally, however, minimum cost adaptation attempts to optimize the system performance with respect to each data block. As a result, the effect of the individual decisions on the overall system performance is left unaccounted for. In some instances, choosing non-minimum cost execution paths for some data blocks may lower the average processing time for all data blocks. For example, consider a system with a fast processor and a slow processor, under which the minimum cost criterion always chooses to use the fast processor. However, under heavy load conditions, shifting a portion of the load to the slower processor would reduce the average processing time of all data blocks. In the next section, we develop a strategy, called optimal system-wide adaptation, which optimizes an overall system performance objective.

5.3 Optimal System-wide Adaptation

Our development of optimal adaptation is based on the optimal routing strategy in communications networks. In this strategy, the routing process is performed for streams of data blocks rather than for each data block individually. The optimization criterion is system-wide congestion, which can be represented as a sum over the congestion levels on each link. The formulation results in a nonlinear optimization problem, which can be solved by a variety of gradient projection-based algorithms [6].

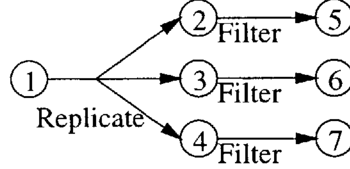
In this section, we also formulate the algorithm adaptation problem as a constrained nonlinear optimization problem. An iterative gradient projection based algorithm is then proposed. A comparison of the performance of this adaptation method and that of minimum cost adaptation is presented in Chapter 9.

5.3.1 Formulation

Recall that our system of processors can be modeled as a collection of virtual processors, each implementing a primitive operation $\mathtt{P} \in \mathcal{P}$, where $\mathcal{P}$ is the set of all primitive operations. Note that there may be more than one processor which implements any particular function.

The processing demands on the system are modeled as a set of N data streams. Data stream i is characterized by an algorithmic expression $\mathtt{A}_\mathtt{i}$, and a production rate λ_i (in data units/s), at which the stream generates new data blocks for processing. Let $\Phi_i = \{\phi_{i1}, \phi_{i2}, \cdots, \phi_{iM_i}\}$ denote

(a)



(b)

Figure 5-3: (a) Multiplier for filtering is 1.75. (b) Multiplier for filtering is 3.

the ordered set of all execution paths for algebraic expression $\texttt{A}_\texttt{i}$. For each $\phi_{ij} \in \Phi_i$, let x_{ij} denote the rate (in data units/s) at which data is processed through execution path ϕ_{ij}, hereafter also referred to as the data flow through path ϕ_{ij}. Finally, $\mathbf{x} = \{x_{ij} | \forall j \in \{1, \ldots, M_i\}, \forall i \in \{1, \ldots, N\}\}$ is the set of unknown variables in the system.

For each execution path ϕ_{ij}, let $M_{ij}(\texttt{P})$ denote the ratio between the load placed on primitive P and the input rate to ϕ_{ij}. $M_{ij}(\texttt{P})$ can be determined with the evaluation of the size() function as described by the rules in Section 5.2.2. For example, in the algorithm graph shown in Figure 5-3(a), the multiplier associated with the filter primitive is 1.75, since the input rate to the second filter is at half the original input rate, and the input rate to the third filter is at a quarter of the original input rate. In Figure 5-3(b), the multiplier associated with filtering is 3, because after the replication by 3, each of the replications need to be filtered. Of course, $M_{ij}(\texttt{P}) = 0$ if the primitive P is not used on path ϕ_{ij}.

Under this setup, the load, or combined input rate, on processors implementing primitive P can be expressed as

$$L_{\texttt{P}}(\mathbf{x}) = \sum_{i,j} M_{ij}(\texttt{P}) x_{ij}. \tag{5.21}$$

The delay, or general cost, associated with processing with primitive P is assumed to be a function of only the load on that class of processors, so that the total cost function is

$$D(\mathbf{x}) = \sum_{\texttt{P}} D_{\texttt{P}}(L_{\texttt{P}}) = \sum_{\texttt{P} \in \mathcal{P}} D_{\texttt{P}} \left[\sum_{i,j} M_{ij}(\texttt{P}) x_{ij} \right], \tag{5.22}$$

where $D_{\texttt{P}}(L_{\texttt{P}})$ is the cost function associated with primitive P.

The optimal execution path assignment problem can now be formulated as follows:

$$\text{minimize} \quad D(\mathbf{x}) = \sum_{\mathrm{P}\in\mathcal{P}} D_{\mathrm{P}}\left[\sum_{i,j} M_{ij}(\mathrm{P})x_{ij}\right] \tag{5.23}$$

$$\text{subject to} \quad \sum_{j=1}^{M_i} x_{ij} = \lambda_i, \qquad \forall i \tag{5.24}$$

$$x_{ij} \geq 0, \qquad \forall j \in \{1,\ldots,M_i\}, \forall i \in \{1,\ldots N\} \tag{5.25}$$

(5.23)–(5.25) expresses a constrained nonlinear optimization problem which minimizes the system cost function (5.23) in terms of the unknown vector of path flows $\mathbf{x}$. The first constraint (5.24) equates the total path flow for a given data stream to the production rate of the data stream. The second constraint (5.25) states that the path flows must be positive.

The optimality conditions for the problem are well-known when $D_{\mathrm{P}}(L_{\mathrm{P}})$ is convex and monotonically increasing with L_{P}. In particular, the path flow vector $\mathbf{x}^*$ is optimal if and only if, for each i

$$x^*_{ij} > 0 \rightarrow \frac{\partial D(\mathbf{x}^*)}{\partial x_{ik}} \geq \frac{\partial D(\mathbf{x}^*)}{\partial x_{ij}}, \qquad \forall j \in \{1,\ldots M_i\}. \tag{5.26}$$

where

$$\frac{\partial D(\mathbf{x})}{\partial x_{ij}} = \sum_{\mathrm{P}\in\mathcal{P}} \frac{dD_{\mathrm{P}}}{dL_{\mathrm{P}}}(L_{\mathrm{P}}(\mathbf{x})M_{ij}(\mathrm{P}). \tag{5.27}$$

These conditions offer some important insights about the form of the optimal solution. First, the processing load is placed on those execution paths with the least marginal cost. In other words, each increment of processing demand is placed on the execution path which causes the least increase in congestion. Second, for each data stream i, more than one path flow x_{ij} may be positive. The implication is that an algorithm adaptation process based on this formulation may cause data blocks produced by the same stream to be processed by different algorithms.

5.3.2 Numerical Solution

A variety of nonlinear programming tools are available for solving (5.23)–(5.25) numerically. Here, we describe how the gradient projection algorithm can be applied to iteratively solve for the optimal path flow vector $\mathbf{x}^*$ [4][5]. To facilitate our discussion, let $\mathbf{x}^k$ denote the estimate of $\mathbf{x}$ at iteration k of the algorithm.

Given $\mathbf{x}^k$, it is possible to rephrase (5.23)–(5.25) to obtain a problem with positivity constraints only. For each data stream i, let ϕ_{q_i} be the path with the minimum first derivative (in the current

iteration), i.e.,

$$\frac{\partial D(\mathbf{x}^k)}{\partial x_{iq_i}} \leq \frac{\partial D(\mathbf{x}^k)}{\partial x_{ij}}, \qquad \forall j \in \{1, \ldots, M_i\} \tag{5.28}$$

Define $\tilde{\mathbf{x}}$ to be the vector of all path flows except those on ϕ_{q_i}. Since

$$x_{iq_i} = \lambda_i - \sum_{j \neq q_i} x_{ij}, \tag{5.29}$$

the cost function can be expressed in terms of $\tilde{\mathbf{x}}$ as

$$\tilde{D}(\tilde{\mathbf{x}}) = \sum_{\mathrm{P} \in \mathcal{P}} D_{\mathrm{P}} \left[\sum_i \left(\sum_{j \neq q_i} (M_{ij}(\mathrm{P}) - M_{iq_i}(\mathrm{P})) x_{ij} + M_{iq_i}(\mathrm{P}) \lambda_i \right) \right] \tag{5.30}$$

(5.23)–(5.25) can now be rephrased as

$$\begin{aligned} &\text{minimize} \quad \tilde{D}(\tilde{\mathbf{x}}) \\ &\text{subject to} \quad \tilde{\mathbf{x}} \geq 0 \end{aligned} \tag{5.31}$$

The application of the scaled gradient projection algorithm to (5.31) is now quite straightforward, since the projection process requires only that non-negative components be set to 0. In other words,

$$x_{ij}^{k+1} = \begin{cases} \max \left\{ 0, x_{ij}^k - \alpha_k \left[\frac{\partial^2 \tilde{D}(\tilde{\mathbf{x}}^k)}{\partial x_{ij}^2} \right]^{-1} \frac{\partial \tilde{D}(\tilde{\mathbf{x}}^k)}{\partial x_{ij}} \right\}, & j \neq q_i \\ \lambda_i - \sum_{j \neq q_i} x_{ij}^{k+1} & j = q_i \end{cases} \tag{5.32}$$

where α_k is a step-size variable,

$$\begin{aligned} \frac{\partial \tilde{D}(\tilde{\mathbf{x}}^k)}{\partial x_{ij}} &= \sum_{\mathrm{P}} (M_{ij}(\mathrm{P}) - M_{iq_i}(\mathrm{P})) \frac{dD_{\mathrm{P}}(L_{\mathrm{P}})}{dL_{\mathrm{P}}} && (5.33) \\ &= \frac{\partial D(\mathbf{x}^k)}{\partial x_{ij}} - \frac{\partial D(\mathbf{x}^k)}{\partial x_{iq_i}}, && (5.34) \end{aligned}$$

and

$$\frac{\partial^2 \tilde{D}(\tilde{\mathbf{x}})}{\partial x_{ij}^2} = \sum_{\mathrm{P}} (M_{ij}(\mathrm{P}) - M_{iq_i}(\mathrm{P}))^2 \frac{d^2 D_{\mathrm{P}}(L_{\mathrm{P}})}{dL_{\mathrm{P}}^2} \tag{5.35}$$

(5.32) represents an iteration of a diagonally-scaled gradient projection method. The scaling coefficient given by (5.35) is guaranteed to be positive due to the convexity of cost functions $D_{\mathrm{P}}(L_{\mathrm{P}})$.

It is chosen so that overall scaling is a diagonal inverse Hessian approximation to $\nabla^2 \tilde{D}$.

Generally, the difficulty in a literal application of a generic gradient projection algorithm to the constrained minimization problem of Section 5.3.1 is that the number of possible execution paths is typically far too many to enumerate explicitly. The algorithm presented here avoids this problem by allowing the possibility of dynamic generation of execution paths. To see this, notice that (5.32) cannot increase the flow on any execution path with non-minimal first derivative cost. Therefore, if $x_{ij}^k = 0$, and ϕ_{ij} is not the minimal first derivative cost path, $x_{ij}^{k+1} = 0$. In fact, in any iteration, the only path flow which can be increased is the minimum first derivative cost path ϕ_{iq_i}. As a result, the system needs to keep track of only *active* paths, which are those with positive path flow. This property is further clarified in Section 7.3.3, where we discuss the implementation details of the algorithm.

5.3.3 Discussion

Because of its basis on an overall system objective, optimal algorithm adaptation does not suffer from the shortcomings of shortest path routing. It is stable, and capable of achieving the maximum throughput of any algorithm expression. Moreover, the convergence properties of the application of the gradient projection algorithm are well-established.

Its possible disadvantages lie in implementational restrictions. Unlike minimum cost algorithm adaptation, which can be applied to either entire data streams or to every block individually, optimal adaptation must be applied to data streams. The significance of this restriction depends on the nature of the processing. However, a large class of signal processing applications naturally require the segmentation of a data stream into blocks prior to processing. A simple example is the overlap-add or overlap-save methods of convolution.

5.4 Summary

The development of algorithm adaptation is motivated by many considerations which suggest that in dynamic and heterogeneous environments, the task of algorithm optimization or selection is better accomplished by the system at runtime, rather than by the user. The system has better knowledge of its own state, and has control over many concurrent jobs, allowing it to perform algorithm selection more judiciously. System transparency is better supported when the user is freed from considerations of implementational details. Finally, system performance can be improved because algorithm adaptation is a resource allocation strategy which adapts to current system congestion levels.

Two different algorithm adaptation strategies are presented in this chapter. Based on shortest path routing in data communications networks, minimum cost packet-wise adaptation can be im-

plemented by recursively applying a set of rules to the algebraic expression attached to each data block individually. While this strategy is straight-forward to understand and implement, it is not able to fully take advantage of all of system processing capacity, and it is susceptible to unstable behavior. Optimal system-wide adaptation overcomes these disadvantages, by using a system-wide cost function as the objective function of a constrained optimization problem which is solved to obtain the execution path allocations for streams of data blocks. The numerical solution of the problem can be performed via an application of a gradient projection method.

Chapter 6

System Capacity

The previous chapter explores the problem of adapting the processing requirements of each data block to the state of a given system of processors. This chapter explores a few issues related to the throughput, or capacity, of a set of processors with respect to a set of processing demands. First, given a system of specialized processors, what is the theoretical upper bound to the throughput of the system with respect to a set of algorithms? An analysis into this question offers insights on the performance of any algorithm adaptation technique. Section 6.2 defines measures of system capacity with respect to single and multiple algorithms. Section 6.3 then builds on these measures to study the question of how a system should be designed to guarantee levels of capacity. Both sets of formulations rely on the key idea of Section 6.1, namely, that data flow in a valid algorithm graph can be equivalently decomposed into either arc flows or path flows.

6.1 Flow Decomposition

The "flow" of data in an algorithm graph refers to the rate (in data units/s) at which data is processed through different parts of the graph. This term is defined when the demand on the system is modeled by data streams, which send out new data blocks for processing according to some statistical process. We envision these output blocks of the data stream as "flowing" through the arcs and paths of the algorithm graph. Generally speaking, the data flow through a given set of arcs and nodes refers to the rate at which data *enters* this set.

One way to define flow is as in Section 5.3, where the flow of data through an execution path is defined as the rate at which data passes through the set of nodes and arcs comprising the execution path. To be more precise, the flow on an execution path refers to the rate at which data *enters* the origination node of the path. Because a typical graph consists of many execution paths, the overall flow on the graph is described by the flows on each of the paths. The use of an *execution path flow decomposition* is intuitive in the context of algorithm adaptation, because every data block must

be assigned to an execution path.

A second way to define flow is to focus on each arc independently, attributing a flow across each arc of the algorithm graph to describe the overall flow. An arc flow is more precisely defined as the rate at which data enters an arc. Because each arc of the graph is associated with a primitive function, such a expression of the flow allows us to quickly ascertain the amount of system resources utilized to support the flow. As we discuss this chapter, the use of an *arc flow decomposition* is more natural in the context of capacity measurement, where the focus is on ensuring that no resource is used beyond its capacity.

In this section, we show a well-defined set of conditions under which these two flow decomposition are entirely equivalent. That is, under these conditions, flows through execution paths can be decomposed into arc flows and vice versa. With this result, the choice of the decomposition becomes a matter of convenience rather than necessity, simplifying many formulations for capacity measurements later in the chapter.

6.1.1 Notation

For convenience, we first restate the following notations from previous chapters. An algorithm graph $\mathcal{G}$ is represented by its set of nodes $\mathcal{N} = \{1, 2, ..., N\}$ and arcs $\mathcal{A}$. The origin and destination nodes are always numbered 1 and N, respectively. Each arc $a \in \mathcal{A}$ can be written as (T_a, H_a, p_a), where T_a represents the ordered set of tail nodes of the arc, H_a represents the ordered set of head nodes of the arc, and p_a is the processor type associated with the arc. Let $|T_a|$ and $|H_a|$ denote the cardinality of the sets T_a and H_a, respectively. For each node n, let $I_n = \{a \in \mathcal{A} | n \in H_a\}$ and $O_n = \{a \in \mathcal{A} | n \in T_a\}$. I_n and O_n are the sets of incoming and outgoing arcs at node n, respectively. In this chapter, we assume that the graph is valid according to the conditions of Section 4.3.6.

As in Chapter 5, for each arc a with associated primitive function $\mathsf{P} = p_a$, we define the following:

- k_a: the node number of the reference input of arc a.
- S_P: the input data size transition vector with respect to the reference input. That is, if Ψ_{T_a} is the vector of input data sizes, and Ψ_{k_a} is the data block size at the reference input, then

$$\Psi_{T_a} = S_\mathsf{P} \Psi_{k_a}.$$

- T_P: the output data size transition vector of P with respect to the reference input. That is, if Ψ_{H_a} is the vector of output data sizes, and Ψ_{k_a} is the data size at the reference input, then

$$\Psi_{H_a} = T_\mathsf{P} \Psi_{k_a}.$$

Additionally, for $\mathcal{G}$, let $S_\mathsf{P} = \{a | p_a = \mathsf{P}\}$ be the set of arcs labeled with primitive P.

Let $\Phi = \{\phi_1, \phi_2, \dots, \phi_M\}$ denote the ordered set of all execution paths in an algorithm graph $\mathcal{G}$. A flow decomposition onto execution paths associates each $\phi_i \in \Phi$ with the rate x_i at which data *enters* the execution path (in data units/s). Let $\mathbf{x}$ be the "path flow" vector of all of the variables $\{x_1, \dots, x_M\}$.

On the other hand, an arc flow decomposition associates a flow with each arc. Because arcs in algorithm graphs have multiple head and tail nodes, we introduce a slightly more complicated set of variables. For each arc a, let $y_{t,a}$ represent the input rate from tail node $t \in T_a$ into arc a and $y_{a,h}$ represent the output rate from the arc a into the head node $h \in H_a$. The resulting set of variables $\{y_{t,a}, y_{a,h}\}$ describes the rates at which data flows from each tail node into each arc, and from each arc into its head nodes.

Let $\mathbf{y}$ represent the "arc flow" vector of all variables $\{y_{t,a}, y_{a,h}\}$ in the graph $\mathcal{G}$. For convenience, if v is a vector of node numbers, $y_{v,a}$ represents the corresponding vector of flows from those nodes in v into arc a. Similarly, $y_{a,v}$ represents the vector of flows from arc a into those nodes in v. A valid $\mathbf{y}$ satisfies the following constraints:

$$
\begin{aligned}
y_{T_a,a} &= S_{p_a} y_{k_a,a} & \forall a \in \mathcal{A} \qquad (6.1)\\
y_{a,H_a} &= T_{p_a} y_{k_a,a}, & \forall a \in \mathcal{A} \qquad (6.2)\\
\sum_{a \in I_n} y_{a,n} - \sum_{a \in O_n} y_{n,a} &= 0, & n \neq 1, N \qquad (6.3)\\
y_{t,a} &\geq 0, & \forall t \in T_a, \forall a \in \mathcal{A} \qquad (6.4)\\
y_{a,h} &\geq 0, & \forall h \in H_a, \forall a \in \mathcal{A} \qquad (6.5)
\end{aligned}
$$

(6.1) and (6.2) applies the definitions of the input and output data size transition vectors to all arcs in the graph. (6.3) expresses conservation of data flows at all internal nodes. In other words, the sum of data flow into a node must equal the sum of data flows out of the node. Finally, (6.4) and (6.5) are positivity constraints.

In the following, we show that any path flow vector can be represented by a unique valid arc flow vector. Conversely, in a graph with no interdependent nodes, any valid arc flow vector can be represented by a (not necessarily unique) path flow vector. This is essentially a parallel statement of the Flow Decomposition Theorem which exists for standard graphs [33].

6.1.2 Path Flow to Arc Flow Conversion

Based on the properties of execution paths in Section 4.3.5, any execution path flow $\mathbf{x}$ can be specified by a unique valid arc flow $\mathbf{y}$. To see this, consider first the arc flow resulting from a unit path flow on a single execution path ϕ_i.

An execution path ϕ_i contains a subset of the nodes and arcs in $\mathcal{G}$, and has the property that

it contains no decision nodes. This property enables us to associate a flow variable f_n with each node n in ϕ_i, representing the rate at which data enters and leaves node n. Since node n has a single incoming arc a_1 and a single outgoing arc a_2, $y_{a_1,n} = f_n$ and $y_{n,a_2} = f_n$. The calculation of f_n is particular simple. If a node n is not in ϕ_i, $f_n = 0$. Otherwise, after setting $f_1 = 1$, we merely need to use S_{p_a} and T_{p_a} to compute f_n for the remaining nodes.

After calculating $\{f_n, \forall n\}$, $\mathbf{y}^i$, the arc flow vector associated with a single unit of data flow on execution path ϕ_i is found with a simple conversion:

$$y^i_{n,a} = \begin{cases} f_n & a \in \phi_i, n \in T_a \\ 0 & \text{otherwise;} \end{cases} \tag{6.6}$$

and

$$y^i_{a,n} = \begin{cases} f_n & a \in \phi_i, n \in H_a \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

It can be easily verified that the resulting $\mathbf{y}^i$ satisfies (6.1)–(6.5).

More generally, with a set of path flows specified by $\mathbf{x}$, $\mathbf{y}$ can be obtained as a simple linear combination

$$\mathbf{y} = \sum_i x_i \mathbf{y}^i \tag{6.8}$$

The resulting $\mathbf{y}$ satisfies (6.1)–(6.5) as a result of the linearity of the operations.

6.1.3 Arc Flow to Path Flow

Consider now the reverse problem of mapping an arc flow $\mathbf{y}$ to a set of execution path flows $\mathbf{x}$. We propose an iterative procedure which deducts the arc flows associated with some execution path from $\mathbf{y}$ in each iteration.

The success of the procedure rests on a single observation: Given a non-zero arc flow vector $\mathbf{y}$ which satisfies (6.1)–(6.5), it is possible to select an execution path ϕ such that every arc in ϕ has non-zero flow with the following algorithm:

1. $A := \{1\}$

2. Find an arc a such that

    - $y_{n,a} > 0$, for all $n \in T_a$; and
    - $T_a \subset A$

3. $A := A \setminus T_a \cup H_a$. Add a and H_a to the execution path ϕ. If $A = \{N\}$, stop; else repeat step 2.

In this algorithm A represents the set of nodes with no outgoing arcs thus far in the construction of ϕ. In Step 2, the first condition is satisfied by every node in A due to the conservation of flow condition (6.3). The graph validity conditions of Section 4.3.6 guarantee that the conditions in the second step are satisfied by at least one node in A, and that this algorithm terminates successfully.

Since the resulting execution path is clearly a member of Φ, let us assume that $\phi = \phi_i$. The arc flow resulting from a single unit of data flow on ϕ_i is $\mathbf{y}^i$, found in our previous discussions. Letting

$$x_i = \min_{a \in \phi_i, n = k_a} \frac{y_{n,a}}{y^i_{n,a}}$$

we have found the path flow on one execution path. Furthermore, suppose a new set of arc flows is defined as

$$\mathbf{y}' := \mathbf{y} - x_i \mathbf{y}^i$$

Due to linearity, $\mathbf{y}'$ still satisfies (6.1)–(6.5). Moreover, the definition of x_i ensures that, compared with $\mathbf{y}$, $\mathbf{y}'$ contains at least one more arc with zero flow. This process of constructing another execution path with nonzero flow can be repeated on $\mathbf{y}'$. The iteration of the entire process guarantees that at least one arc flow is reduced to 0 in each iteration, so that the decomposition would be complete in at most $|\mathcal{A}|$ iterations.

The significance of the mapping between arc flows and path flows is that it allows us to formulate problems in terms of whichever set of flow variables is more convenient. Path flows are intuitively well-matched to the concept that each data block is processed by a single, complete execution path. For this reason, the optimal algorithm adaptation strategy of Chapter 5 is described in terms of path flows. However, the explicit enumeration of all possible execution paths in a graph is extremely cumbersome, while the enumeration of arc flows is quite simple. Moreover, arc flows make clear the amount of "load" carried by each arc, and hence the virtual processors of each type. Therefore arc flows are better suited for investigating the capacity of the system with respect to the graph.

The decomposition result of this section guarantees that any arc flow satisfying (6.1)–(6.5) represents a valid set of path flows, and is therefore achievable via an algorithm adaptation scheme. For this reason, in the discussions of this chapter, the constraints (6.1)–(6.5) are abbreviated as

$$\mathcal{N}\mathbf{y} = 0 \tag{6.9}$$

$$\mathbf{y} \geq 0 \tag{6.10}$$

The matrix $\mathcal{N}$ is a characteristic of the algorithm graph, and is analogous to the flow transition

matrices in conventional directed graphs.

6.2 System Capacity

In discussing flow decomposition, the data flow through the algorithm graph is viewed from a purely theoretical point of view. However, the flow of data through a graph is in fact not a theoretical or mathematical process, but rather a transformational process which requires work by the designated class of primitive operators of each arc. At the same time, like any physical system, the speed of individual processors is limited, and therefore the total speed of processors of a given type is limited. Hence for any algorithm graph, there exists an upper bound to the rate at which the system can "transmit," or process, data through that graph. The subject of this section is a characterization and study of this upper bound, also called the system capacity. Generally speaking, the system capacity refers to the maximum rate at which the system can process the data labeled with an algorithm graph without exceeding the processing rate limits of the processors in the system.

Section 6.2.2 presents a mathematical definition of the system capacity with respect to a single algorithm graph. In the absence of any other processing demands, when the input rate into an algorithm graph is at or below the system capacity, there exists an execution path allocation which guarantees that the processing limits of every primitive operator is respected. On the other hand, when the input rate exceeds the system capacity, every execution path allocation exceeds the processing limits of some primitive operator.

Building on the definition of single algorithm capacity, Section 6.2.3 studies the definition of the system capacity with respect to multiple algorithm graphs. Instead of the single algorithm capacity, we study the set of input rates at which the system can simultaneously satisfy the processing demands of a set of data streams. This set, called the achievable rate region, is mathematically described, and we show that the region is convex and bounded by hyperplanes.

6.2.1 System Model

As stated in Chapter 3, the system is modeled as a collection of virtual processors, each having a single functionality $\mathtt{P} \in \mathcal{P}$. Each primitive $\mathtt{P}$ can therefore be characterized by the $C_{\mathtt{P}}$, the total processing capacity of virtual processor of type $\mathtt{P}$. $C_{\mathtt{P}}$, also called the capacity of primitive $\mathtt{P}$, intuitively refers to the maximum rate at which the system can handle requests to use primitive $\mathtt{P}$. If each processor can be modeled as an $M/M/1$ queue, for example, when the input rate exceeds the capacity, the queue size is no longer bounded. Let $\mathbf{C}$ denote the vector of all primitive capacities $C_{\mathtt{P}}$.

The processing demands on the system are modeled as a set of N data streams. Each data stream is characterized by an algorithmic expression $\mathtt{A}$, and a production rate λ (in data units/s),

at which the stream generates new data blocks for processing. Again, we make the simplifying assumption that the cost of processing a data block scales linearly with the size of the data block.

6.2.2 Single Algorithm Capacity

We begin by defining the system capacity with respect to a single data stream with algorithm $\mathcal{A}$. Intuitively, the *single algorithm capacity* of the system is the maximum rate at which the system can process data through algorithm graph $\mathcal{A}$, in the case that all system resources are dedicated for this task. Mathematically, it is a function of the algorithm and the capacity of each primitive in the system, and can be defined as

$$\mathfrak{C}(\mathcal{A}, \mathbf{C}) = \max_{\mathbf{y}} \sum_{a \in O_1} y_{1,a} \tag{6.11}$$

$$\text{subject to} \quad \sum_{a \in S_{\mathsf{P}}} \sum_{t \in T_a} y_{t,a} \leq C_{\mathsf{P}}, \qquad \forall \mathsf{P} \in \mathcal{P} \tag{6.12}$$

$$\mathcal{N}\mathbf{y} = 0 \tag{6.13}$$

$$\mathbf{y} \geq 0 \tag{6.14}$$

In words, the capacity of the system is the maximum input rate into the algorithm graph for which there exists a valid arc flow whose demand on primitive functions does not exceed their associated capacities. (6.11) states that the sum of flows out of the origination node of $\mathcal{A}$ is maximized over the set of arc flows. (6.13) and (6.14) restricts the optimization to be over valid arc flows only. (6.12) restricts the sum of input rates at all arcs requiring the primitive P to less than C_{P}.

To facilitate further discussions, we introduce additional notation so that (6.11)–(6.14) can be more conveniently expressed in the following matrix form:

$$\mathfrak{C}(\mathcal{A}, \mathbf{C}) = \max_{\mathbf{y}} \mathbf{f}^T \mathbf{y} \tag{6.15}$$

$$\text{subject to} \quad \mathcal{B}\mathbf{y} \leq \mathbf{C} \tag{6.16}$$

$$\mathcal{N}\mathbf{y} = 0 \tag{6.17}$$

$$\mathbf{y} \geq 0 \tag{6.18}$$

$\mathbf{f}$, called the *input capacity vector* of $\mathcal{A}$, is a vector of the same length as $\mathbf{y}$, so that

$$\mathbf{f}^T \mathbf{y} = \sum_{a \in O_1} y_{1,a}.$$

Moreover, The matrices $\mathcal{B}$ and $\mathcal{N}$ are called the *capacity characteristic matrices* of the algorithm graph $\mathcal{A}$.

The optimization problem (6.15)–(6.18) can be solved using any linear programming algorithm.

Let $\mathbf{y}^*$ denote the (not necessarily unique) arc flow which optimizes (6.15). By the flow decomposition result of the previous section, $\mathbf{y}^*$ can be decomposed into a set of execution path flows. Hence, if the production rate of the data stream is below $\mathfrak{C}(\mathcal{A}, \mathbf{C})$, there exists at least one execution path allocation which does not exceed the processing capacity of any primitive. In other words, any processing rate below $\mathfrak{C}(\mathcal{A}, \mathbf{C})$ is *achievable* by the system. If the output rate of the data stream exceeds $\mathfrak{C}(\mathcal{A}, \mathbf{C})$, then every execution path allocation exceeds the processing capacity of at least one primitive.

$\mathbf{y}^*$ also reveals the "bottlenecks" of the system. Specifically, those primitives $\mathtt{P}$ for which (6.16) is satisfied with strict equality are the bottlenecks of the system. The capacity of the system can be increased by increasing the processing capacity $C_{\mathtt{P}}$ of those primitives. Additionally, if $\mathbf{y}^*$ is not unique, every solution reveals a set of primitives whose processing capacity limits the overall system capacity with respect to the given algorithm.

The solution of (6.15)–(6.18) does not reveal *how* data blocks should be allocated onto execution paths. However, it does provide a benchmark against which algorithm adaptation strategies can be measured. For example, a static implementation of the minimum cost adaptation strategy of Section 5.2 cannot achieve the capacity limit except in the simplest of the graphs. The reason is that it assigns only one execution path to all of the data blocks in the data stream, while the system may require the use of multiple execution paths to handle high output rates. The system-wide optimal adaptation strategy of Section 5.3 is theoretically capable of achieving throughput up to the system capacity, provided that the appropriate cost functions are used.

6.2.3 Multiple Algorithm Capacity

Realistically, however, a system must handle more than one data stream simultaneously. When the same primitive operator appears in more than one algorithm graph, the system capacities with respect to those graphs are interdependent. Therefore, a more appropriate description of the system capacity is the set of simultaneously achievable processing rates. The members of this set are vectors of rates at which the system can process data through corresponding algorithm graphs without exceeding the capacities of its primitive operators.

Consider a system supporting the processing requests of M data streams, with algorithm graphs $\mathcal{A}_i$ for $i \in \{1, 2, \ldots, M\}$. Each graph $\mathcal{A}_i$ is associated with capacity characteristic matrices $\mathcal{N}_i$ and $\mathcal{B}_i$, as well as the input capacity vector $\mathbf{f}_i$. The achievable set of simultaneous processing rates is the set of *rate vectors* $[\lambda_1, \lambda_2, \ldots, \lambda_M]$ for which there exists arc flows $\mathbf{y}_i$ satisfying the following

system of equations:

$$\begin{aligned}
\lambda_i &= \mathbf{f}_i^T \mathbf{y}_i & i \in \{1, \ldots, M\} \qquad (6.19)\\
\sum_{i=1}^{M} \mathcal{B}_i \mathbf{y}_i &\leq \mathbf{C} & (6.20)\\
\mathcal{N}_i \mathbf{y}_i &= 0 & i \in \{1, \ldots, M\} \qquad (6.21)\\
\mathbf{y}_i &\geq 0 & i \in \{1, \ldots, M\} \qquad (6.22)
\end{aligned}$$

Intuitively, if a rate vector satisfies (6.19)–(6.22), then the system can support valid arc flows for every algorithm graph without exceeding the capacity of any primitive operator. This set of vectors satisfying (6.19)–(6.22) is denoted $\mathfrak{C}([\mathcal{A}_1, \ldots, \mathcal{A}_M], \mathbf{C})$.

The achievable set of simultaneous processing rates clearly depends on the algorithm graphs themselves. However, some characteristics apply to all achievable sets. As a simple example, it is bounded by the hyperplanes

$$\lambda_i \geq 0, \quad i \in \{1, \ldots, M\}; \qquad (6.23)$$

$$\lambda_i \leq \mathfrak{C}(\mathcal{A}_i, \mathbf{C}), \quad i \in \{1, \ldots, M\}. \qquad (6.24)$$

(6.23) is always satisfied due to the positivity constraints (6.22). (6.24) holds due to the consideration that the simultaneous processing rates can never exceed the single algorithm capacity of any algorithm.

More generally, the achievable set can be shown to be a convex set bounded by hyperplanes. In other words, the achievable rate region is a convex polytope, which is fully described by its corner points. The observation that it is bounded by hyperplanes is a simple consequence of the linearity of (6.19) and (6.22).

To show its convexity, consider two achievable rate vectors α and β, both of which are in $\mathfrak{C}([\mathcal{A}_1, \ldots, \mathcal{A}_M], \mathbf{C})$. They can be associated with valid arc flows $\mathbf{y}^\alpha$ and $\mathbf{y}^\beta$, satisfying (6.20)–(6.22). Consider now a rate vector

$$\gamma = s\alpha + (1 - s)\beta,$$

where $0 \leq s \leq 1$. Let

$$\mathbf{y}^\gamma = s\mathbf{y}^\alpha + (1 - s)\mathbf{y}^\beta.$$

For each $i \in \{1, \dots, M\}$,

$$\gamma_i = s\alpha_i + (1-s)\beta_i = s\mathbf{f}_i^T \mathbf{y}^\alpha + (1-s)\mathbf{f}_i^T \mathbf{y}^\beta = \mathbf{f}_i^T \mathbf{y}^\gamma. \tag{6.25}$$

Furthermore,

$$\sum_{i=1}^{M} \mathcal{B}_i \mathbf{y}_i^\gamma = \sum_{i=1}^{M} \mathcal{B}_i (s\mathbf{y}_i^\alpha + (1-s)\mathbf{y}_i^\beta) = s\sum_{i=1}^{M} \mathcal{B}_i \mathbf{y}_i^\alpha + (1-s)\sum_{i=1}^{M} \mathcal{B}_i \mathbf{y}_i^\beta \leq \mathbf{C} \tag{6.26}$$

Additionally, for each $i \in \{1, \dots, M\}$,

$$\mathcal{N}_i \mathbf{y}_i^\gamma = \mathcal{N}_i (s\mathbf{y}_i^\alpha + (1-s)\mathbf{y}_i^\beta) = 0 \tag{6.27}$$

And finally,

$$\mathbf{y}^\gamma \geq 0. \tag{6.28}$$

The combination of (6.25)–(6.28) clearly parallels the conditions of (6.19)–(6.22). Since γ is in the achievable rate region, $\mathfrak{C}([\mathcal{A}_1, \dots, \mathcal{A}_M], \mathbf{C})$ is convex.

To gain a more intuitive understanding of the achievable rate region, we examine some examples for the $M = 2$ case. Suppose that a given system consists of 4 primitives `A`,`B`, `C`, and `D`, and the primitive capacity vector is

$$\mathbf{C} = \begin{bmatrix} C_{\mathtt{A}} \\ C_{\mathtt{B}} \\ C_{\mathtt{C}} \\ C_{\mathtt{D}} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Figure 6-1 shows a table of sets of algorithm graphs and their associated achievable rate region in this system. In all 4 examples, the achievable rate region is convex and bounded by lines.

In the top example, the two algorithm graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ use no common primitive functions, and hence the system capacities with respect to the two graphs are independent. As a result, the achievable rate region is the rectangular region bounded by the single algorithm capacities of the individual algorithms.

The second example shows a case in which the same primitive appears in the only execution paths of both graphs. Because the execution of both algorithms require the use of primitive `B`, the achievable rate region is bounded by the constraint:

$$\lambda_1 + \lambda_2 \leq C_{\mathtt{B}} = 1.$$

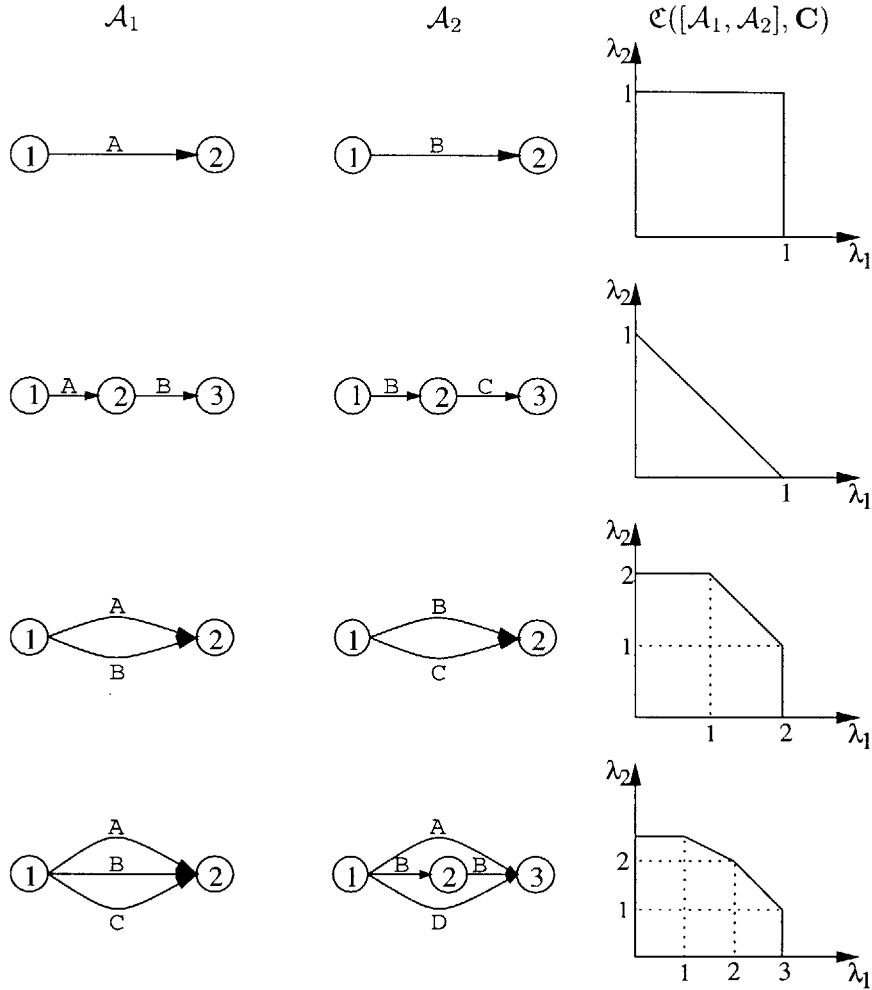


Figure 6-1: Examples of the achievable rate region for four different sets of algorithms.

In the third example, the achievable rate region reflects the fact that the two algorithms both use primitive B in one of their execution paths. The achievable rate pair $[1, 2]$ corresponds to an arc flow of 1 on primitive A in $\mathcal{A}_1$, along with flows of 1 on both primitives B and C in $\mathcal{A}_2$. On the other hand, the achievable rate pair $[2, 1]$ corresponds to arc flows of 1 on both primitives A and B in $\mathcal{A}_1$, along with flow of 1 on primitives C in $\mathcal{A}_2$.

In the last and most complex example, both algorithm graphs contain 3 execution paths. In $\mathcal{B}_2$, the primitive B is used twice in the "middle" execution path. The outer boundary of achievable rate region consists of 3 "corner" points. To achieve the corner point $[1, 2.5]$, the system puts a flow of 1 on primitive C in $\mathcal{A}_1$. This allows flows of 1 on arcs A and D and flows of 0.5 on the two B arcs in $\mathcal{A}_2$. Similarly, to achieve the corner point $[2, 2]$, the system puts arc flows of 1 on primitives B and C in $\mathcal{A}_1$, leaving flows of 1 on arcs A and D on $\mathcal{A}_2$. Finally, the corner point $[3, 1]$ is achieved with arcs flows of 1 on primitives A, B, and C on $\mathcal{A}_1$, and an arc flow of 1 on arc D on $\mathcal{A}_2$.

6.3 System Design with Capacity Considerations

For a given system and a given set of processing demands, the calculation of the system capacity determines whether the processing demands can indeed be satisfied. If the demands cannot be met, how can the primitive capacities in the system be adjusted to increase the capacity? Even if the demands can be met, can the number of processors in the system be reduced while still meeting the capacity requirements? These are questions faced by the system designer with the task of assigning primitive functions to processors in environments such as those built on DSP network chips, described in Section 3.3.2.

In such environments, the system designer is faced with requirements that the system capacity with respect to a given set of algorithm graphs meets or exceeds some minimum. Her design variables are $K_{\mathtt{P}}$, the number of processors assigned to primitive function P, called the primitive allocation of P for short. Every processor assigned to function P operates at the same operating rate $R_{\mathtt{P}}$, so that the primitive capacity for primitive P is

$$C_{\mathtt{P}} = R_{\mathtt{P}} K_{\mathtt{P}}.$$

Letting $\mathbf{K}$ denote the vector of processor assignments $K_{\mathtt{P}}$, the vector of primitive capacities can expressed as

$$\mathbf{C} = \mathcal{R}\mathbf{K}, \tag{6.29}$$

where $\mathcal{R}$ is a diagonal matrix whose non-zero entries correspond to $R_{\mathtt{P}}$. This section formulates the problem of choosing $K_{\mathtt{P}}$ to minimize the total number of required processors while meeting the capacity requirements of three different kinds.

First, in Section 6.3.1, we study the simple case where the capacity requirement is with respect to a single algorithm graph. Section 6.3.2 then the formulates the problem where capacity requirement is with respect to multiple algorithm graphs simultaneously. Finally, Section 6.3.3 deals with the case in which the processing demands are uncertain.

6.3.1 Single Algorithm Case

We first consider the design specification that the system capacity with respect to a given algorithm graph $\mathcal{A}$ is at least λ. The algorithm graph is characterized by its input capacity vector $\mathbf{f}$ and capacity characteristic matrices $\mathcal{B}$ and $\mathcal{N}$.

The design of interest is one requiring the least number of processors. Mathematically, it can

be obtained by solving the following integer programming problem:

$$\min_{\mathbf{K},\mathbf{y}} \quad \sum_{\mathrm{P}\in\mathcal{P}} K_{\mathrm{P}} \tag{6.30}$$

$$\text{subject to} \quad \mathbf{f}^T\mathbf{y} \geq \lambda \tag{6.31}$$

$$\mathcal{N}\mathbf{y} = 0 \tag{6.32}$$

$$\mathcal{B}\mathbf{y} \leq \mathcal{R}\mathbf{K} \tag{6.33}$$

$$\mathbf{y} \geq 0 \tag{6.34}$$

$$K_{\mathrm{P}} \in \{0, 1, \dots\} \quad \mathrm{P} \in \mathcal{P} \tag{6.35}$$

The optimization is performed over $\mathbf{K}$, the vector of primitive allocations K_{P}, and $\mathbf{y}$, the vector of arc flows over the algorithm graph. The objective function (6.30) is simply the total number of processors in the system. (6.31) expresses the constraint that the system capacity exceeds λ, while (6.32)–(6.34) restrict the optimization to be over valid arc flows only. Finally, (6.35) restricts the processor assignments to be integral.

Under the integrality constraint, the result of the optimization problem (6.30)–(6.35) is not easily characterizable. However, it is interesting to consider the linear programming relaxation of the problem, obtained by disregarding the integrality constraint (6.35). In the following, we show that the relaxed problem has a simple and intuitive solution, based on finding the most hardware efficient execution path.

As in Section 6.1.1, let $\Phi = \{\phi_1, \phi_2, \dots, \phi_M\}$ denote the ordered set of all execution paths in an algorithm graph $\mathcal{G}$. For each path ϕ_i, let $\mathbf{y}^i$ represent the the arc flow vector associated with a single unit of data flow on execution path ϕ_i. The (not necessarily integral) primitive allocation required to support the flow $\mathbf{y}^i$ is

$$K_{\mathrm{P}}^i = \frac{1}{R_{\mathrm{P}}} \sum_{\{(n,a) | p_a = \mathrm{P}, n \in T_a\}} y_{n,a}^i,$$

for all primitives P. The most hardware efficient execution path ϕ_{i^*} is the one requiring the lowest total primitive allocation to support its unit arc flow vector, i.e.,

$$i^* = \arg\min_i \sum_{\mathrm{P}} K_{\mathrm{P}}^i. \tag{6.36}$$

Let $\mathbf{K}^*$ denote the primitive allocation associated with execution path i^*. Notice that for any graph, there may be more than one most hardware efficient execution path.

With the application of the flow decomposition theorem, we see that the solution to the relaxed optimization problem of (6.30)–(6.34) is simply $\lambda\mathbf{K}^*$. Recall that any valid arc flow can be decom-

posed into execution path flows. Therefore, any arc flow distribution requiring the use of execution paths other than ϕ_{i^*} would require more hardware resources per unit of input flow. As a result, the optimal solution is to use only the most hardware efficient execution path(s).

In summary, if the integrality constraints can be relaxed, the system should simply concentrate its resources on the execution path requiring the least amount of resources. This design strategy mirrors common algorithm design practice today; for a given task, the designer simply chooses the one most efficient algorithm, and processes all of the data with this algorithm. The problem and solution are analogous to that of designing a communication network for the sole purpose of facilitating communications between two given locations. If hardware cost and communications capacity are the only considerations, it is reasonable to simply place a dedicated connection of the lowest cost between the two nodes of interest.

In both scenarios, the resulting system design allows only one path between the origin and destination nodes, thus precluding the use of algorithm adaptation (in the context of processing) and routing (in the context of communications). This result is largely due to the fact that the design specification involves only one origin-destination node pair, and contains no provisions for possible sources of change or uncertainty in the processing or routing demand. The system design is indeed optimal if no other algorithm graph is used, or no other location requires communications. However, unpredicted changes in the user demand may require major changes in the system design.

6.3.2 Simultaneous Capacity Requirements

Consider now a more complex situation in which the system must handle multiple data streams simultaneously. As in Section 6.2.3, the system supports M data streams, with algorithm graphs $\mathcal{A}_i$ for $i \in \{1, 2, \ldots, M\}$. Each graph $\mathcal{A}_i$ is associated with capacity characteristic matrices $\mathcal{N}_i$ and $\mathcal{B}_i$, as well as the input capacity vector $\mathbf{f}_i$. The design specification requires that the rate vector $[\gamma_1, \gamma_2, \ldots, \gamma_M]$ is a member of the achievable rate region $\mathfrak{C}([\mathcal{A}_1, \ldots, \mathcal{A}_M], \mathbf{C})$.

Again the design of interest is the one requiring the fewest number of processors. The mathematical formulation is the following integer programming problem:

$$\min_{\mathbf{K}, \mathbf{y}_1, \ldots, \mathbf{y}_M} \quad \sum_{\mathrm{P} \in \mathcal{P}} K_{\mathrm{P}} \tag{6.37}$$

$$\text{subject to} \quad \mathbf{f}_i^T \mathbf{y}_i \geq \gamma_i \quad \forall i \tag{6.38}$$

$$\mathcal{N}_i \mathbf{y}_i = 0 \quad \forall i \tag{6.39}$$

$$\sum_{i=1}^{M} \mathcal{B}_i \mathbf{y}_i \leq \mathcal{R} \mathbf{K} \tag{6.40}$$

$$\mathbf{y}_i \geq 0 \quad \forall i \tag{6.41}$$

$$K_{\mathrm{P}} \in \{0, 1, \ldots\} \quad \mathrm{P} \in \mathcal{P} \tag{6.42}$$

As in the single algorithm case, the optimization is performed with free variables $\mathbf{K}$, the vector of primitive allocations $K_{\mathtt{P}}$, and $\mathbf{y}_i$, the arc flow vectors over each algorithm graph. The only difference between the formulations is that the primitive capacity must exceed the usage over all of the algorithms, as expressed by (6.40).

Even though the system must now support multiple data streams, the solution of the linear programming relaxation problem (6.37)–(6.41) is a simple extension of that for the single algorithm case. Specifically, the design criterion leads us to allocate processors only on the most hardware efficient execution path(s) of each algorithm graph. Essentially, if there is any graph for which the path flows are not concentrated on the most hardware efficient execution path, the hardware cost can be reduced by switching all flows to the most efficient path. Despite the fact that multiple algorithm graphs are considered, the design specifications still contain no provisions for possible sources of change or uncertainty in the processing demand.

Because multiple algorithm graphs are involved, however, it is interesting to examine the achievable rate region of a system designed with the formulation of (6.37)–(6.42). Due to the convexity property of the achievable rate region, $\mathfrak{C}([\mathcal{A}_1, \ldots, \mathcal{A}_M], \mathbf{C})$ contains the set of points $(\lambda_1, \ldots, \lambda_M)$ for which

$$0 \leq \lambda_1 \leq \gamma_1 \tag{6.43}$$

$$0 \leq \lambda_2 \leq \gamma_2 \tag{6.44}$$

$$\ldots \tag{6.45}$$

$$0 \leq \lambda_M \leq \gamma_M. \tag{6.46}$$

Because each graph contains many possible execution paths, however, the resulting achievable rate region may actually encompass a significantly larger area.

Consider two simple two-algorithm design examples with three possible primitive functions, A, B, and C, whose associated operating rates are

$$R_{\mathtt{A}} = 1; \qquad R_{\mathtt{B}} = \frac{1}{2}; \qquad R_{\mathtt{C}} = \frac{1}{3}$$

Figure 6-2 shows the primitive allocation $\mathbf{K}$ and the resultant achievable rate region for two different sets of algorithm graphs. In each example, the design specification is that $[\gamma_1, \gamma_2] = [1, 1]$ is in the achievable rate region.

In the top example of Figure 6-2, the two algorithm graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ are singleton arcs. To ensure $\gamma_1 = 1$, we allocate $K_{\mathtt{A}}$ to be 1; to ensure $\gamma_2 = 1$, we allocate $K_{\mathtt{B}}$ to be 2. With this primitive allocation, the achievable rate region is simply the square region with the corners at the origin and $[\gamma_1, \gamma_2] = [1, 1]$.

In the bottom example of Figure 6-2, the two algorithm graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ both contain two

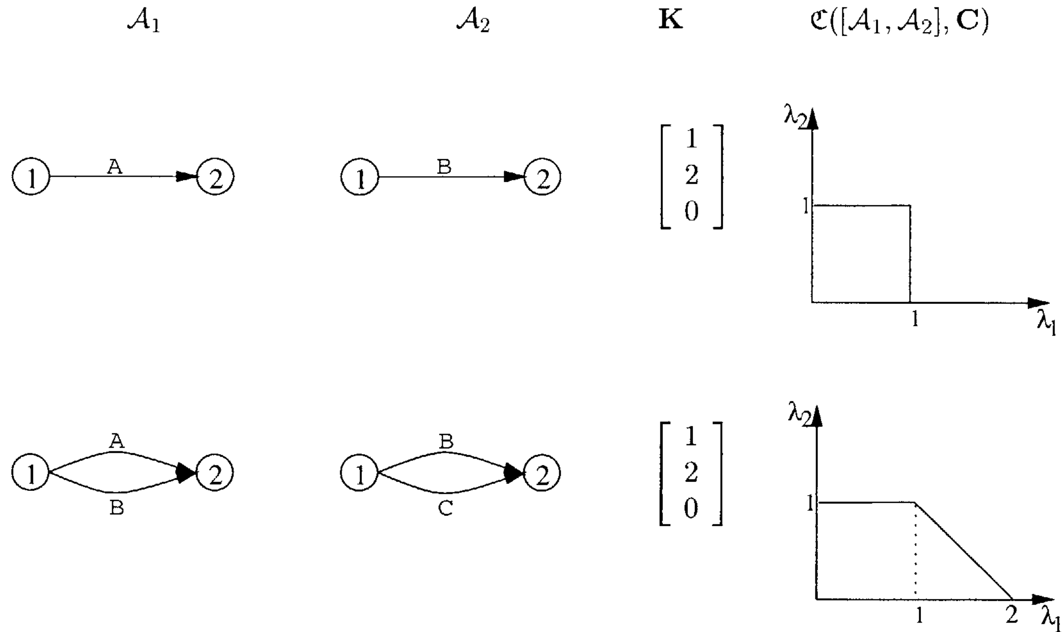


Figure 6-2: Examples of the primitive allocation and achievable rate region for two sets of algorithms.

execution paths. For $\mathcal{A}_1$, the most hardware efficient execution path is the arc A, and hence $K_{\mathtt{A}}$ is assigned 1. For $\mathcal{A}_2$, the most hardware efficient execution path is the arc B, and hence $K_{\mathtt{B}}$ is assigned 2. It is in fact the same primitive allocation as the previous example. However, because $\mathcal{A}_1$ can use both A and B, the point $[\lambda_1, \lambda_2] = [2, 0]$ is a corner point of the achievable rate region. It is achieved when all of the system resources are used for processing $\mathcal{A}_1$.

6.3.3 Uncertain Capacity Requirements

In the previous sections, the system design specifications do not account for the elements of uncertainty in modeling the demand. As a result, the solutions are simple, but amenable to changes in demand. In this section, we address a case in which the demand on the system is not completely known ahead of design time. Consider again a system supporting M data streams. Data stream i outputs data blocks labeled with algorithm graph $\mathcal{A}_i$ at rate γ_i, and algorithm graph $\mathcal{A}_i$ is associated with capacity characteristic matrices $\mathcal{N}_i$ and $\mathcal{B}_i$, as well as the input capacity vector $\mathbf{f}_i$. This time, the design specification requires that the system be able to support *any* of these data streams, one at a time. In other words, all of the following points are in the achievable rate region

of the system $\mathfrak{C}([\mathcal{A}_1, \dots, \mathcal{A}_M], \mathbf{C})$:

$$[\gamma_1, 0, 0, \dots, 0]$$
$$[0, \gamma_2, 0, \dots, 0]$$
$$\dots$$
$$[0, 0, 0, \dots, \gamma_M]$$

This design specification models the situation in which the required processing task is either unknown or dynamically changing, but the system must be able to handle all forms of processing demands.

The design of interest is the one requiring the fewest number of processors, found by solving the following integer programming problem:

$$\min_{\mathbf{K}, \mathbf{y}_1, \dots, \mathbf{y}_M} \quad \sum_{\mathrm{P} \in \mathcal{P}} K_{\mathrm{P}} \tag{6.47}$$
$$\text{subject to} \quad \mathbf{f}_i^T \mathbf{y}_i \geq \gamma_i \quad \forall i \tag{6.48}$$
$$\mathcal{N}_i \mathbf{y}_i = 0 \quad \forall i \tag{6.49}$$
$$\mathcal{B}_i \mathbf{y}_i \leq \mathcal{R}\mathbf{K} \quad \forall i \tag{6.50}$$
$$\mathbf{y}_i \geq 0 \quad \forall i \tag{6.51}$$
$$K_{\mathrm{P}} \in \{0, 1, \dots\} \quad \mathrm{P} \in \mathcal{P} \tag{6.52}$$

Just as in the previous section, the formulation of (6.47)–(6.52) performs a minimization of the total number of required processors with free variables $\mathbf{K}$, the vector of primitive allocations K_{P}, and $\mathbf{y}_i$, the arc flow vectors over each algorithm graph. However, unlike (6.40), (6.50) specifies that the primitive capacity must exceed the usage over each individual graph.

The solution of (6.47)–(6.52) is the smallest set of processors which guarantees that the single algorithm capacity of the system with respect to algorithm $\mathcal{A}_i$ is γ_i. The resulting system design is capable of handling any of the M data streams, one at a time. In addition, due to convexity, the multiple algorithm achievable rate region contains the set of points $[\lambda_1, \lambda_2, \dots, \lambda_M]$ which can be expressed as a convex combination of the corner points of the region. In other words, if there exists $\{\alpha_1, \dots, \alpha_M\}$ such that

$$\lambda_i = \alpha_i \gamma_i \quad i \in \{1, \dots, M\} \tag{6.53}$$
$$\sum_{i=1}^{M} \alpha_i = 1 \tag{6.54}$$
$$\alpha_i \geq 0 \tag{6.55}$$

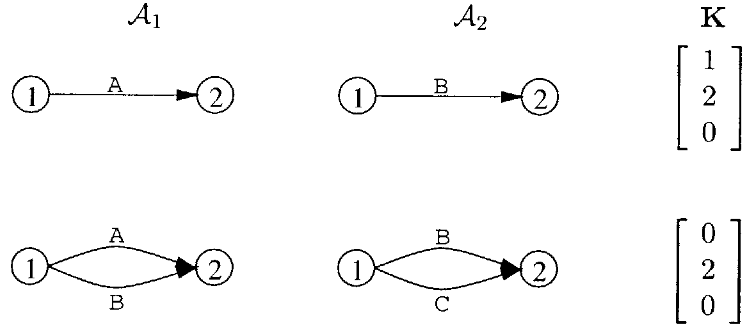


Figure 6-3: Examples of the primitive allocation for two sets of algorithms.

then $[\lambda_1, \lambda_2, \dots, \lambda_M]$ is in the achievable rate region of the system.

To illustrate this design criterion, we now examine several two-algorithm design examples with three possible primitive functions, A, B, and C, whose associated operating rates are

$$R_{\mathtt{A}} = 1; \qquad R_{\mathtt{B}} = \frac{1}{2}; \qquad R_{\mathtt{C}} = \frac{1}{3}$$

Figure 6-3 shows the primitive allocation **K** for two sets of algorithm graphs. In each example, the design specification is that $\gamma_1 = 1$ and $\gamma_2 = 2$.

In the top example of Figure 6-3, the two algorithm graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ are singleton arcs. Because they use no common primitives, the primitive allocations are the same as in the previous section. To ensure $\gamma_1 = 1$, we allocate $K_{\mathtt{A}}$ to be 1; to ensure $\gamma_2 = 1$, we allocate $K_{\mathtt{B}}$ to be 2.

In the second example, the two algorithm graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ both contain two execution paths. Allocating two processors to primitive B is enough to guarantee that the single algorithm capacities of the system with respect to both algorithms is 1. In the previous section, the design specification favors allocating primitives only on the most hardware efficient path of the algorithms; in this section, however, the design specifications favor designs which take advantage of overlaps in the algorithmic expressions. Sets of algorithms with overlaps can be implemented with fewer processors than those sets without overlaps.

6.4 Summary

Conceptually, the system capacity of a given set of processors with respect to a given algorithm is the maximum rate at which data can be processed without exceeding the processing capacity of the individual processors. To facilitate the mathematical formulation of this idea, the equivalence of path flows and arc flows is first established. Then, the system capacity with respect to a given algorithm graph is formulated using arc flow variables, and the achievable rate region of the system

with respect to multiple algorithms is shown to be a convex set bounded by hyperplanes.

The capacity measure is independent of the algorithm adaptation strategy, and hence serves as an benchmark of the relative merits of different strategies. Its solution reveals the bottleneck of the system with respect to the given processing tasks. Also, accurate knowledge of the system capacity permits the system to control the release of new tasks onto the network under heavy congestion. In other words, if it is known that the system capacity would be exceeded with the addition of a new task, then the system should prevent or delay its entry into the network.

Using our definition of system capacity, we study the problem of functionality assignment to a set of processors in order to meet a set of processing demands. When all processing demands are known and constant, processors should be assigned to primitives involved in the execution path incurring least hardware cost. However, when processing demands are uncertain, the solution highlights the benefits of resource sharing gained in offering the system options on the actual sequence of steps used to process data.

Chapter 7

Simulation Environment

In the previous chapters, we describe a conceptual framework for distributed signal processing. Chapter 3 begins by modeling the system as a collection of virtual specialized processors, while Chapter 5 describes methods for dynamically selecting algorithms for processing data streams according to the current system conditions. Chapter 6 presents theoretical bounds on the overall throughput, or capacity, of systems implementing algorithm adaptation. In the next three chapters, we take a more practical approach, and assess the actual performance of the proposed algorithm adaptation schemes via simulation. This chapter describes the simulation environment and discusses issues encountered in implementing algorithm adaptation.

7.1 Objects and Input/Output

The simulation environment emulates a distributed environment of sensors and processors implementing algorithm adaptation in which the resource of interest is time. It is an event-based simulation program written in an object-oriented language, so that the resulting design can be viewed as a collection of autonomous, interacting objects. The set of objects is chosen to reflect our system model that the processing network is a collection of specialized virtual processors, and that data objects are blocks of data tagged with a description of the required processing steps. In this section, we overview the basic objects in the simulation and specify the inputs and outputs of the simulation environment. Descriptions of the interactions among the objects are left to the next section.

7.1.1 Input

The inputs of the simulation environment are the following:

- The functionality and processing rate of each processor.

- The data production rate and algebraic expression of the processing task of each sensor.
- The rate of the communications channel between every pair of sensors and processors.
- Specification of the type of algorithm adaptation: packet-wise or system-wide.
- Duration of the simulation.

7.1.2 Network

Based on the inputs, the simulation creates a network consisting of nodes and links between the nodes. Sensors and processors are the two different types of network nodes.

Characterized by its data production rate and associated algorithm description, every sensor manages a single data stream. The data stream produces data block objects of a fixed size according to a Poisson process with a pre-specified rate. If system-wide optimization is used, the sensor is also responsible for deriving the execution path flow variables for the data stream according to the iterative algorithm described in Section 5.3.2.

The second type of network node is a specialized processor, characterized by its functionality, processing rate, failure rate, and recovery rate. Each processor maintains a queue for the data blocks requiring its service. During processing, it modifies the data block, as well as the algorithm description attached to it. Upon completion, the processor examines the algorithm expression and sends the data block to the next appropriate network node.

The communications network connecting the network nodes is characterized by the transmission speeds between every pair of nodes in the network. Rather than explicitly simulating the routing functions of an underlying communications network, we assume that between every pair of network nodes there exists a "virtual" link of pre-specified capacity. This structure reflects the assumption in Chapter 3 that communications and control costs are minimal within this study.

7.1.3 Data Blocks

Each data block in the simulation environment encapsulates the following information:

- Its current size: The size of the data block changes as a result of processing, and its current size affects the time required for the current processing step.
- Its processing requirements: Each data block is labeled with an algebraic description of its processing requirements. This label changes when the current processing step is completed, and when the algorithm adaptation process makes a choice at the decision nodes of the algorithm graph.
- Its final node, which is the network node where it is to be sent when the processing requirements are all fulfilled.

- Its processing history: The times at which this data block enters and exits queues and processors are recorded to facilitate the collection of statistics.

The system procedures for handling data blocks are described in the next section.

7.1.4 Output

During the simulation, the program collects information on the set of data blocks whose processing was completed during pre-specified intervals of time, and presents the following statistics at the end of each interval:

- The number of data blocks produced by each sensor;
- The number of data blocks whose processing was completed;
- The average time between the production of a data block and its completion;
- The average total time spent in queues;
- The average total time spent on processing;
- The number of data blocks processed through each execution path.

By analyzing these statistics, we learn about the system's overall performance in terms of throughput and latency, and study how the execution path selections differ for different algorithm adaptation schemes. In addition, we study how the system performance reacts to changes in the production rate of the sensors and to sudden failures of processors.

7.2 Handling Concurrency

As described in Chapter 4, there are three methods of composition within the algebraic expressions: sequential, concurrent, and alternative. The handling of sequential constructs is fairly intuitive; when a processor finishes the current processing step, it examines the algebraic expression and sends the data block onto the next step. However, both concurrency and equivalence constructs require extra design decisions. This section considers concurrency constructs, which require the disassembly of a data block into its components, and then the assembly of multiple results into a whole.

To properly disassemble data blocks into separate entities, network nodes (both processors and sensors) need to accomplish the following tasks:

- Recognize that the next processing step is a concurrency construct requiring disassembly.
- Perform disassembly correctly, so that each new data block encapsulates the appropriate data, algebraic expression, and destination node. The destination node of the new data blocks is

the network node where the processing results are assembled, and is also called the "assembly node."

- Send out the new data blocks in the network

In the simulation environment, the assembly node is always the current processor or sensor. However, as long as the assembly node is notified that it is responsible for assembling the results of processing, other choices are available. For example, one design is to designate the processor to be used after the concurrency construct has been completed. A second design is to choose a node which is idle and contains large amounts of memory. Other choices may take into account communications and other overhead costs associated with assembly of the processing results.

When the assigned assembly node differs from the current processor, network delays might cause the notification to arrive late. In such cases, the returning data blocks gather at the assembly node prior to a proper notification. Protocols must be designed to handle such occurrences. Our choice of using the current processor has the advantage that the assembly node is instantly notified of its role, so that this never happens.

The chosen destination node must accomplish the following tasks:

1. Create a "placeholder" data block.

2. As data blocks arrive into the node, recognize those which require assembly rather than processing. These data blocks should not enter the same queue as those waiting for processing, because that would introduce unnecessary delays to the assembly process.

3. Match those data blocks requiring assembly with the appropriate placeholder data block.

4. Merge the data as well as the processing records into the placeholder data block.

5. Recognize when the placeholder data block has all of the requisite components, and send it back out onto the network.

In the simulation environment, the process of assembly is assumed to be an instantaneous procedure. This procedure does not involve computation, and hence we assume that it does not consume a significant amount of time or other resources.

7.2.1 Example

To illustrate the implementational issues, the following example traces the system activities necessary to fulfill the processing requirements of a single data packet whose algebraic expression consists of only sequential and concurrency constructs.

Primitive P	Data Size Transition Matrix	Speed
A	$\begin{bmatrix} 1 \\ 1 \end{bmatrix}$	1
B	$\begin{bmatrix} 1.5 & 0 \\ 0 & 0.5 \end{bmatrix}$	2
C	1	3
D	1	4
E	$\begin{bmatrix} 1 & 1 \end{bmatrix}$	5

Figure 7-1: Primitives and associated data size transition matrix.

Consider a network consisting of five primitive operators shown in Figure 7-1, and one sensor which produces data blocks of unity size, labeled with the following algorithmic expression

$$\mathtt{y} = \mathtt{E} \begin{bmatrix} \mathtt{C} \\ \mathtt{D} \end{bmatrix} .* \mathtt{BAx}. \tag{7.1}$$

The destination node of the data blocks is the sensor node itself. This expression does not contain any alternative constructs, so the system activities related to algorithm adaptation are inconsequential with respect to this particular setup.

Suppose that the sensor produces a data block at time 0. This triggers a sequence of events in the network to handle the demands of this data block.

1. First, the data block is sent to processor A. The times at which it enters and leaves the queue of A are stamped. T, the time that the processor spends on this data block, is a random variable following the exponential distribution:

   $$f_T(t) = \begin{cases} \alpha e^{-\alpha t}, & t \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{7.2}$$

   The parameter α is the quotient of the processor speed and the size of the data block, so that in this case, $\alpha = 1$.

   Primitive A is a dual-output operator. However, because both output blocks are inputs to the same primitive function, a "super" data block consisting of two data blocks, each with size 1, is created. The algorithmic expression attached to this super block is

   $$\mathtt{y} = \mathtt{E} \begin{bmatrix} \mathtt{C} \\ \mathtt{D} \end{bmatrix} .* \mathtt{B} \begin{bmatrix} \mathtt{x}_1 \\ \mathtt{x}_2 \end{bmatrix}.$$

2. The super block is transmitted to processor B. The times at which it leaves A and arrives at

the queue of B are stamped. The time that the processor spends on the data block again is an exponentially distributed random variable of the form (7.2), with the parameter $\alpha = 2/2 = 1$.

Primitive B outputs two data blocks. Because the next step is a concurrency construct, the simulation disassembles the output into two different data blocks. The first has size 0.5, and is labelled with algebraic expression

$$\mathtt{y}_1 = \mathtt{C}\mathtt{x}_1.$$

The second has size 1.5, and is labelled with algebraic expression

$$\mathtt{y}_2 = \mathtt{D}\mathtt{x}_2.$$

Both data blocks are assigned the same destination node so that they would be transmitted to the same place when the processing is complete. In the simulation, the convention is to assign the current node, processor B, as the destination node.

After sending both data blocks out to their respective processors, processor B creates a placeholder data block with algebraic expression

$$\mathtt{y} = \mathtt{E}\begin{bmatrix} \mathtt{x}_1 \\ \mathtt{x}_2 \end{bmatrix},$$

and saves it in a storage area for placeholder data blocks.

3. (a) One of the data blocks is sent to processor C, where the processing time is an exponentially distributed random variable of the form (7.2) with parameter $\alpha = 3/1.5 = 2$. Upon completion, the data block has size 1.5, and is sent back to processor B.

   (b) The other data block is sent to processor D. The parameter α for the exponential distribution of the processing time is $\alpha = 4/.5 = 8$. Upon completion, the data block has size 0.5, and is also sent back to processor B.

Because the returning data blocks do not require further processing by processor B, they do not enter the processing queue. Instead, they are immediately matched with the placeholder data block; the data and processing records contained within each are transferred. When they have both returned, the placeholder becomes a super block, with data sizes 1.5 and 0.5, and algebraic expression

$$\mathtt{y} = \mathtt{E}\begin{bmatrix} \mathtt{x}_1 \\ \mathtt{x}_2 \end{bmatrix}.$$

4. The events at processor E are the same as those previously described. Upon completion at processor E, the data block has size 2, and is transmitted to the destination node, which is the sensor in this case. At the sensor, statistics on the processing and queueing times are collected.

7.3 Handling Equivalence

The handling of equivalence constructs requires two interacting processes. The first process associates a cost with each processor or processor type, while the second process updates the decision variables associated with the different alternatives according to the cost evaluations. These processes interact since changes in the routing variables affect the processor loads, and thereby the processor costs. This section describes the implementations of packet-wise minimum cost adaptation and system-wide optimization algorithm adaptation. We begin with a discussion on the how processor costs are evaluated in the system, and then go on to describe the implementation decisions made for each algorithm adaptation technique.

7.3.1 Cost Function

Physically, the utilization cost for applying a given processor or processor type to a given job should reflect the amount of resources dedicated to the processing. While each processing job requires a variety of different resources, the selected cost function should reflect the resource which is most likely to be a bottleneck. Depending on the processor type, some of these resources may be memory, power, bandwidth, etc. For simplicity, in the simulation environment, the cost function is chosen to be the congestion level of the processors, as measured by their average workload.

More specifically, suppose that the network contains $N_{\mathtt{P}}$ processors implementing primitive operation P, and each processor $\mathtt{P}_i$ has the processing rate of $\mu_{\mathtt{P}_i}$ data units/s. The waiting system associated with each processor is modeled as an $M/M/1$ queue. That is, we assume that the arrival process of data blocks is memoryless, and the first-order interarrival time is a random variable following an exponential probability density function. Furthermore, the probability distribution of the service time (i.e., time it takes for the processor to complete the requisite processing for a given data packet) is exponential. Each processor has an infinite queue, so that no data is rejected or dropped by any processor.

In the simulation, for each primitive operation P, the load $L_{\mathtt{P}}$ is distributed among these $N_{\mathtt{P}}$ processors according to their processing rates, so that the probability p_i that a data block is processed by processor $\mathtt{P}_i$, given that it requires processing by primitive P, is

$$p_i = \frac{\mu_{\mathtt{P}_i}}{\sum_{i=1}^{N_{\mathtt{P}}} \mu_{\mathtt{P}_i}}.$$

Physically, a data block requiring processing by $\mathtt{P}$ is assigned to $\mathtt{P}_i$ with probability p_i, regardless of the current load on $\mathtt{P}_i$.

When packet-wise minimum cost adaptation is used, it is necessary to assign a cost $C_{\mathtt{P}}$ for each primitive functionality $\mathtt{P}$. While many different choices exist for this cost function, in the simulation, $C_{\mathtt{P}}$ is chosen to be the average queue length of processors implementing primitive $\mathtt{P}$. More specifically, suppose that the network contains $N_{\mathtt{P}}$ processors implementing primitive operation $\mathtt{P}$, and each processor $\mathtt{P}_i$ has the processing rate of $\mu_{\mathtt{P}_i}$ data units/s and a queue length of $Q_{\mathtt{P}_i}$ data units. Then the cost function is defined as follows:

$$C_{\mathtt{P}} = \frac{\sum_{i=1}^{N_{\mathtt{P}}} Q_{\mathtt{P}_i}}{\sum_{i=1}^{N_{\mathtt{P}}} \mu_{\mathtt{P}_i}} \tag{7.3}$$

To use (7.3) as the cost function, each processor must keep track of the sum of data block sizes in its queue, and broadcast this information to the other nodes in the network periodically. There are of course other choices for the cost function. For example, one can measure the average delay of packets leaving each processor, and vary the cost functions according to priorities of jobs. The choice of (7.3) is due to the simplicity of implementation, as well as reasonable performance observed in the experiments.

When algorithm adaptation via system-wide optimization is used, each primitive $\mathtt{P}$ is associated with a cost function $D_{\mathtt{P}}(L_{\mathtt{P}})$ relating the cost to the load. In the simulation, the cost of $\mathtt{P}$ reflects the expected delay in the queueing systems of processors implementing $\mathtt{P}$:

$$S_{\mathtt{P}}(L_{\mathtt{P}}) = \sum_{i=1}^{N_{\mathtt{P}}} p_i \frac{1}{\mu_{\mathtt{P}_i} - p_i L_{\mathtt{P}}} = \frac{N_{\mathtt{P}}}{\sum_{i=1}^{N_{\mathtt{P}}} \mu_{\mathtt{P}_i} - L_{\mathtt{P}}} = \frac{N_{\mathtt{P}}}{K_{\mathtt{P}} - L_{\mathtt{P}}} \tag{7.4}$$

where $K_{\mathtt{P}}$ is the total processing capacity of type $\mathtt{P}$ processors. The cost function associated with function $\mathtt{P}$, $D_{\mathtt{P}}$, is the following function of the offered load $L_{\mathtt{P}}$:

$$D_{\mathtt{P}}(L_{\mathtt{P}}) = \begin{cases} 0, & L_{\mathtt{P}} \in (-\infty, 0) \\ S_{\mathtt{P}}(L_{\mathtt{P}}), & L_{\mathtt{P}} \in [0, 0.99K_{\mathtt{P}}) \\ S_{\mathtt{P}}(0.99K_{\mathtt{P}}) + S'_{\mathtt{P}}(0.99K_{\mathtt{P}})(L_{\mathtt{P}} - 0.99K_{\mathtt{P}}) & L_{\mathtt{P}} \in [0.99K_{\mathtt{P}}, \infty) \\ \quad + \frac{1}{2}S''_{\mathtt{P}}(0.99K_{\mathtt{P}})(L_{\mathtt{P}} - 0.99K_{\mathtt{P}})^2, & \end{cases} \tag{7.5}$$

(7.5) is a reasonable cost function to be associated with each type of primitive operation in the system. It is convex and monotonically increasing with the load $L_{\mathtt{P}}$, making it possible to use the framework of Section 5.3. Its use requires the estimation of the work load $L_{\mathtt{P}}$ of each functionality $\mathtt{P}$. In the simulation, each processor $\mathtt{P}_i$ estimates the rate $\lambda_{\mathtt{P}_i}$ at which data blocks arrive for its processing. It does so by keeping a running sum of the sizes of the data blocks entering its queue over a fixed interval of time. At the end of each interval, the arrival rate $\lambda_{\mathtt{P}_i}$ is estimated by dividing

this sum with the length of the interval, and the sum is reset to 0. The load λ_{P_i} is then broadcast throughout the network.

In the simulation, all network nodes in the network have access to the same table of the most recent estimates of processor loads or queue sizes when making the algorithm adaptation decisions. Specifically, we do not account for communications delays in broadcasting processor information throughout the network.

Finally, both (7.3) and (7.5) assign costs to groups of processors with the same functionality, rather than to individual processors. Higher levels of specificity is of course possible. Doing so would allow the optimizations to become specific to the conditions of every processor, resulting in better overall system performance at the cost of higher overhead costs. However, in a highly dynamic system, where processors enter and exit the network frequently, the grouping of processors by functionality makes the optimization process less susceptible to abrupt changes in processor availability.

7.3.2 Implementation of Packet-wise Minimum Cost Adaptation

The practical implementation of the packet-wise minimum cost adaptation strategy involves two design considerations: the choice of the cost function and the timing of the execution path selection updates. The rules for finding the minimum cost execution path in any algebraic expression are listed in Section 5.2. The cost per unit data size $C_{\mathtt{P}}$ for using primitive function P is as defined in (7.3).

A second design consideration is the time and frequency of execution path selection. First, each data block can be assigned the minimum cost execution path at the time of its creation. This choice effectively delegates the task of finding the minimum cost path to the sensors only. Because the minimization process requires the evaluation of costs of every branch, it is more efficient to do this all at once. A second option delays every decision until the processing reaches a decision point, so that every processor must be capable of finding the minimum cost execution path. With this choice, the path costs are evaluated with the most recent load information, at the cost of higher computational overhead. In the simulation, the second of these options is used, in order to better accomodate the dynamic nature of the environment.

7.3.3 Implementation of System-wide Optimization

A gradient projection algorithm to solve for the execution path allocation which minimizes system congestion is described in Section 5.3.2. Because it is an iterative algorithm which governs how the packets from the same data stream are distributed, its execution is performed at the sensor. In this section, we describe the implementation of this algorithm, from the perspective of a sensor which produces data blocks attached to algorithm $\mathcal{A}$ at rate λ.

To execute the algorithm described by (5.32), each sensor keeps track of the following variables:

- N^k: the number of active execution paths in iteration k.
- $\Psi^k = \{\psi_i^k, i = 1, ..., N^k\}$: The ordered set of active execution paths in iteration k.
- $\{x_i^k, i = 1, ..., N^k\}$: x_i^k is the path flow variable associated with execution path ψ_i in iteration k.

These variables are initialized as follows:

- $N^0 := 1$.
- $\psi_1^0 :=$ the minimum first derivative cost execution path in the current system state.
- $x_1^0 := \lambda$.

At regular intervals of time, the sensor updates these variables via the following operations:

1. Compute the minimum first derivative cost execution path based on the table of processor loads. Call this path ψ^*.

2. Update the flow variables associated with the non-minimum cost paths which are currently active. For each $i \in \{1, ..., N^k\}$, if $\psi_i^k \neq \psi^*$,

$$x_j^{k+1} = \max\left\{0, x_j^k - \alpha_k \left[\frac{\partial^2 \tilde{D}(\tilde{\mathbf{x}}^k)}{\partial x_j^2}\right]^{-1} \frac{\partial \tilde{D}(\tilde{\mathbf{x}}^k)}{\partial x_j}\right\}. \tag{7.6}$$

   In the simulations, α_k is chosen to be 0.1. Let $\Upsilon = \{\psi_i^k | x_i^{k+1} = 0\}$ be the set of execution paths which have become non-active due to this update.

3. Update the set of active execution paths:

$$\Psi^{k+1} = \Psi^k \cup \{\psi^*\} \setminus \Upsilon. \tag{7.7}$$

   Likewise, update the elements of $\{x_j^{k+1}\}$ to contain the corresponding path flows, and set $N^{k+1} = |\Psi^{k+1}|$.

4. Calculate the flow variable associated with the minimum cost path ψ^*. Suppose $\psi_q^{k+1} = \psi^*$. Then

$$x_q^{k+1} = \lambda - \sum_{i \neq q} x_i^{k+1} \tag{7.8}$$

Processor P	Functionality	Speed	Processor P	Functionality	Speed
1	A	100	6	C	100
2	A	250	7	C	100
3	B	80	8	D	160
4	B	200	9	D	60
5	B	150	10	E	100

Figure 7-2: Processors in the simulation environment.

The above procedure is performed at each sensor in regular intervals of time. If this interval is too long, the convergence time becomes too slow. If the interval is too short, however, the calculations may be based on processor loads which are not accurately estimated. In the simulation, the length of the interval is chosen to be directly proportional to the production rate of the sensor.

7.4 Simple Experiments

In this section, we demonstrate the workings of the simulation environment through a sequence of simple examples. Chapter 9 contains experimental results for a more complex and realistic signal processing task involving the processing of synthetic aperture radar data.

7.4.1 Simulated Network

In this section, the simulated network consists of 10 processors with functionality and speed shown in Figure 7-2. For simplicity, all of the primitive functions are single-input single-output operators which do not change the size of the data blocks. There are two sensors in the network. Sensor 1 produces data blocks labeled with the algorithm description

$$\mathtt{y} = (\mathtt{BA} + \mathtt{DC} + \mathtt{E})\mathtt{x}, \tag{7.9}$$

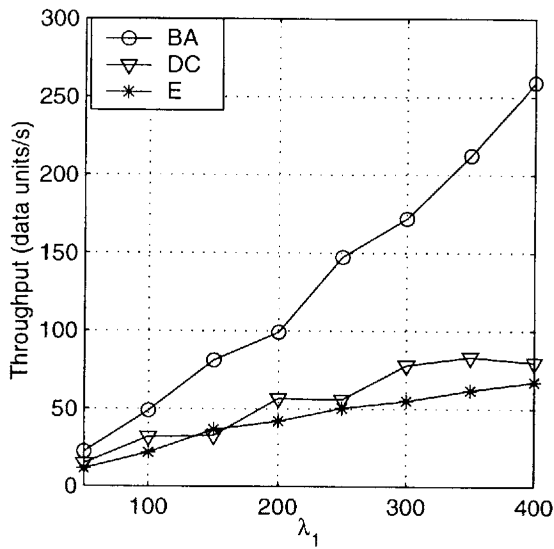
while Sensor 2 produces data blocks labeled with the algorithm descripiton

$$\mathtt{y} = (\mathtt{E} + \mathtt{D})\mathtt{Bx}. \tag{7.10}$$

The production rate of the former is called λ_1, while the production rate of the latter is called λ_2. In the following experiments, λ_2 is fixed at 150, while data points are taken at varying values of λ_1.

7.4.2 Steady-State Allocations

We begin by looking at how the different algorithm adaptation strategies assign execution paths to data blocks as the production rate of Sensor 1 varies.



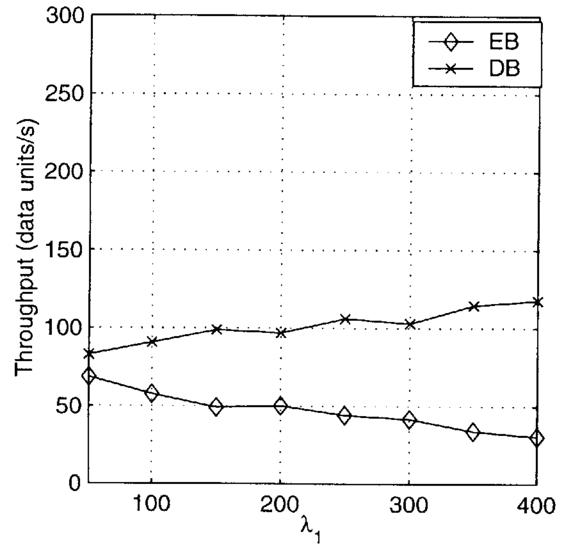
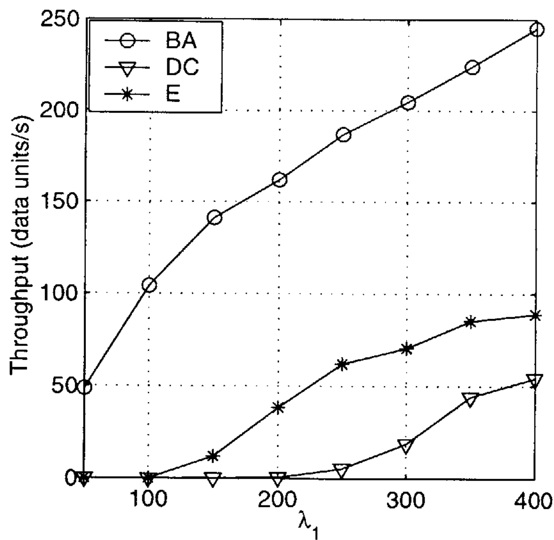


Figure 7-3: Execution path allocations using packet-wise optimization.



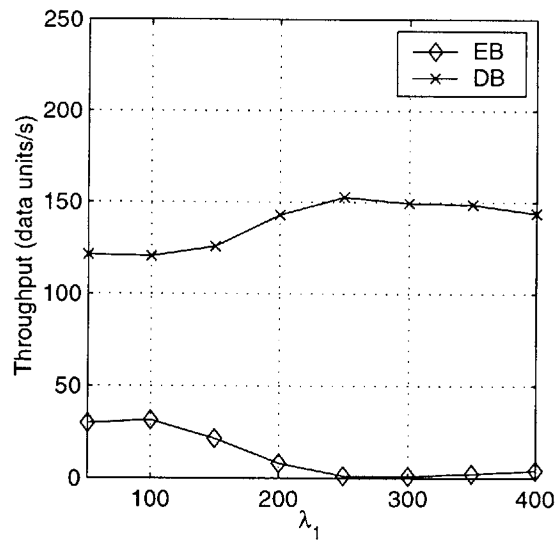


Figure 7-4: Execution path allocations using system optimization.

Figure 7-3 shows the execution path allocations for data blocks from Sensors 1 and 2 when algorithm adaptation is performed by packet-wise optimization. The left graph plots the data block allocations for sensor 1, while the right graph plots the allocations for sensor 2. Each line in each graph corresponds to the rate or throughput at which the labeled execution path outputs data blocks. For instance, when $\lambda_1 = 200$ data units/s, approximately 50% (100 units/s) of the sensor 1 data blocks were processed through the execution path $\mathtt{y} = \mathtt{BAx}$; about 28% were processed through the execution path $\mathtt{y} = \mathtt{DCx}$; and the rest were processed through execution path $\mathtt{y} = \mathtt{Ex}$. At the same time, approximately 67% (100 units/s) of the sensor 2 data blocks were processed through the execution path $\mathtt{y} = \mathtt{DBx}$, while the rest were processed through execution path $\mathtt{y} = \mathtt{EBx}$.

In comparison, Figure 7-4 shows the execution path allocations when algorithm adaptation is performed by system-wide optimization. Again, the left graph plots the data block allocations for sensor 1, while the right graph plots the allocations for sensor 2. One implementational concern for algorithm adaptation via system-wide optimization is the convergence of the path flow vectors. For this simple example, we have verified that the allocations reached during the simulation do indeed converge to the optimal solutions reached offline using MATLAB. Notice that the allocations differs significantly from those in Figure 7-3.

Given that the different adaptation strategies result in different allocation, it is now interesting to evaluate their relative performance. Figure 7-5 plots the average packet delays for data blocks from sensors 1 and 2 and compares them to a baseline case in which no dynamic adaptation is used for data blocks produced by Sensor 1. In this baseline case, the algorithm expression associated with sensor 1 is changed to

$$\mathtt{y} = \mathtt{BAx}, \tag{7.11}$$

corresponding to the situation in which the user pre-selects an algorithm. However, it should be noted that execution path selection is performed via system optimization for data blocks produced by sensor 2. In this scenario, the only processor sharing between the two types of data blocks is for processors implementing function B.

The left graph plots the average packet delay for data blocks from sensor 1 for the three scenarios, while the right graph plots the average packet delay for data blocks from sensor 2. For both sensors, algorithm adaptation via system optimization resulted in the lowest packet delay. Using no algorithm adaptation for sensor 1 resulted in near optimal performance for small λ_1. As the production rate λ_1 increases, however, using no algorithm adaptation leads to a sharp increase in packet delay. Notice from Figure 7-2 that the processing capacity of primitive B is limited to 430 units/s. Because processing of sensor 2 data blocks utilizes 150 units/s, only 280 units/s are available for processor sensor 1 data. Hence it is not surprising that when $\lambda_1 = 300$, the system delays grew unbounded. Finally, packet-wise adaptation is noticeably less effective than system-

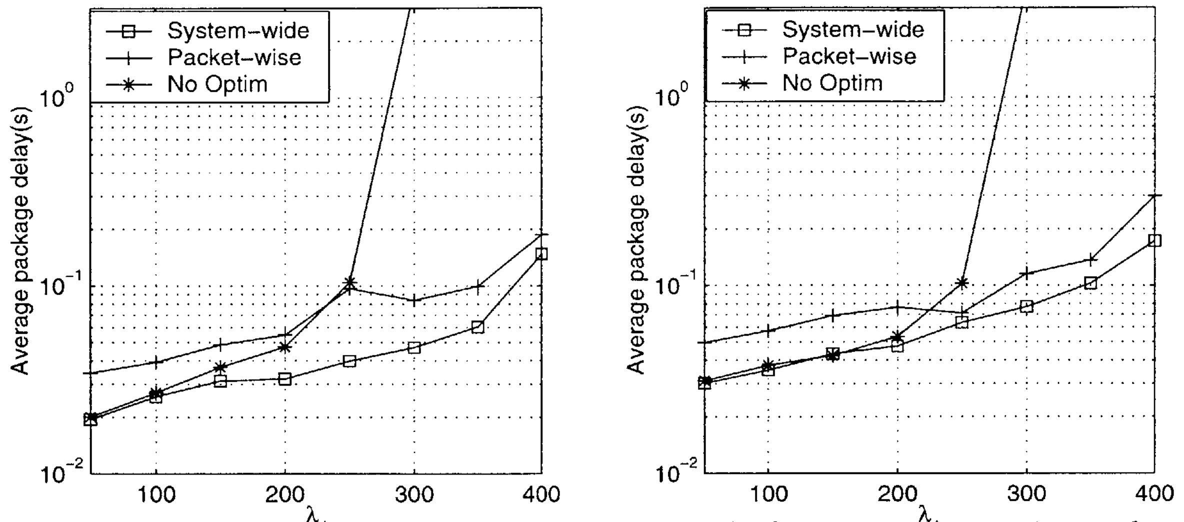


Figure 7-5: Average packet delay for packets in sensors 1 and 2 for system optimization, packet-wise optimization, and no optimization.

wide optimization. However, both adaptation schemes are able to limit packet delay for a broad range of values of λ_1.

7.5 Summary

The primary purpose of the simulation environment is to study the performance and limitations of algorithm adaptation in a dynamic and heterogeneous distributed environment. The details of its construction are significant in helping us gain practical insights into many implementational issues, such as the encapsulation of necessary information within each data packet, the logistics of disassembly/assembly of packets, and the selection and evaluation of cost functions. This chapter describes the resolution of many of these issues within the simulation environment. A simple example with predictable results is used to verify the correctness of the implementation of algorithm adaptation.

Chapter 8

Algorithms for Synthetic Aperture Radar

The previous chapter describes the structure and implementation of our simulation environment. The next chapter uses the simulation to evaluate the performance of algorithm adaptation on the task of image formation for stripmap synthetic aperture radar (SAR) data. SAR represents an interesting and challenging problem for distributed signal processing because it is a highly computationally intensive task. Moreover, as described in this chapter, there exists a large variety of implementational choices for SAR image formation algorithms. This chapter provides an introduction to SAR, along with a discussion on notions of functional equivalence in the context of SAR.

In a conventional radar system, the resolution is inversely proportional to the diameter of the antenna aperture. Thus, the high resolution imaging of a geographic area would require the use of an impractically large antenna. SAR systems, on the other hand, achieve high resolution by combining sophisticated signal processing with a small movable antenna. In SAR, the radar is mounted on a movable platform such as an airplane or a satellite, and a sequence of closely spaced pulses is transmitted and the return waveforms recorded, in effect "synthesizing" a large antenna aperture.

This chapter offers an overview of the field and a survey of image formation algorithms for stripmap mode SAR imaging, in which the antenna is pointed at a fixed direction relative to the flight path, and collects data on a long strip of terrain. The image formation problem can be viewed as one of signal reconstruction: Given the return signals, construct the 2-dimensional space of reflectors which generated the returns. In Section 8.1, we present the fundamental terminology, assumptions, and notations related to general radar imaging. Then, Section 8.2 describes the problem of range resolution, which involves resolving targets in the direction along the line of sight of the radar. Section 8.3 studies the imaging situation in stripmap SAR and offers analytical derivations

of three different stripmap imaging algorithms. Building on these algorithms, Section 8.4 sketches the forms of functional equivalences which are available due to the many ways of implementing various subcomponents of the algorithms.

8.1 Fundamentals of Radar Imaging

This section prepares the reader for the following discussion of the SAR algorithms by presenting fundamental ideas and terminology on radar imaging. Common assumptions and conventions used throughout the chapter are also described.

8.1.1 Microwave imaging

SAR is an active microwave imaging system, generally using radars operating at carrier frequencies of 1-10 GHz, although present day foliage penetrating (FOPEN) and ground penetrating (GPEN) SARs operate in the lower UHF/VHF ranges of nearly 0.3 GHz. The operating frequencies are usually chosen to facilitate all-weather imaging, allowing high-quality reconnaissance through clouds and at night. Additionally, the bandwidth of the transmitted signals are chosen to guarantee certain levels of range resolution and are often on the order of 200 MHz. Microwave signals propagate at the speed of light c.

Microwave imaging measures the microwave reflectivity of objects in the scene. The reflectivity is a complex-valued function which is generally dependent on the frequency of the radar signal and the viewing angle. In most common modes of SAR operation, however, the reflectivity is assumed to not vary significantly over the bandwidth of the signal or over the range of viewing angles. Microwave imaging differs from optical imaging in several ways. First, since the wavelength of microwaves is longer than that of visible light, objects which appear rough on a photograph may show up on SAR images as black silhouettes. Second, radar imaging works on a ranging principle, meaning that objects which differ in elevation but are equidistant in range from the radar map to the same point. This causes an effect known as *layover* whereby elevated objects in the scene becomes displaced in the image. A final difference arises from the fact that SAR utilizes coherent illumination, implying that the reflected signals can add both constructively and destructively, resulting in speckled images [31].

An important fact for algorithm development is that the electric field at the receiver is equal to the linear superposition of electric fields induced by all of the reflectors in the illuminated scene. Hence, one standard technique for developing algorithms uses as a first step the derivation of the echoed waveform due to a single reflector, also called the point target response(PTR). The total response from a scene is the convolution of the PTR with the scene reflectivity function. Therefore, the reconstruction algorithm seeks to "compress" the PTR back to an impulse.

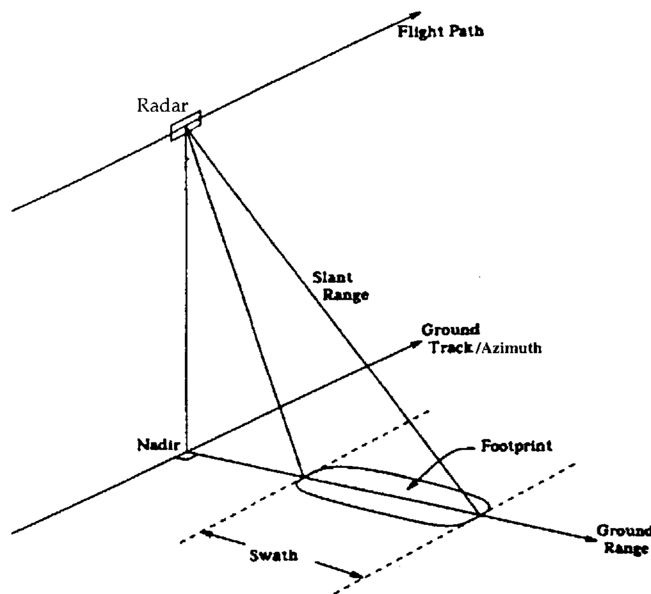


Figure 8-1: Geometry of a side-looking radar system. From [31] with modifications.

8.1.2 Terminology

Consider a side-looking radar antenna pointing down from the air as shown in Figure 8-1. The ground coordinate along the line of sight of the radar is called the *range*, and the ground coordinate perpendicular to the range is termed the *azimuth* or *cross-range*. The *slant range* refers to the radial coordinate measured from the radar rather than from the ground. The *footprint* refers to the area of illumination, the size of which is proportional to the elevation of the radar, and inversely proportional to radar size.

For a conventional radar, two targets on the ground in range X and separated by ΔY in the azimuth direction can be resolved only if they are not in the footprint at the same time. Since the size of the footprint is inversely proportional to the length D of the antenna, the azimuth resolution is

$$\Delta Y = \lambda X/D, \tag{8.1}$$

where λ refers to the operating wavelength of the transmitted energy. Therefore, improved resolution comes with a price of a larger antenna which is more difficult to deploy in space.

The notable insight associated with SAR is that positioning of the radar affects the phase of the received echoes. Therefore, the transmission and reception of multiple pulses from varying locations offer more accurate information about the azimuth location of the targets. Hence the

sophistication associated with SAR algorithms generally concerns only azimuth resolution. In fact, range resolution techniques are the same for both conventional radar and SAR. They are described in Section 8.2, leaving descriptions of the azimuth resolution techniques to Section 8.3.

8.1.3 Assumptions

Because of the limited scope of this chapter, we make a number of simplifying assumptions, including the following:

- The scene being imaged is assumed to be essentially flat, so that the scene reflectivity function is justifiably two-dimensional.
- The returned radar echoes experience the same amount of attenuation regardless of the position of the reflector in the scene.
- Our algorithm developments are mostly done with continuous time or continuous space variables, even though the actual implementations may be through discrete time or discrete space samples. We assume that the signals are appropriately sampled so that the desired mathematical operations are properly implemented.
- Since the velocity of the moving platform is small compared to the speed of light, we assume that between transmission and reception of a single radar pulse, the radar remains in the same position.
- The radar platform moves in a straight path. In other words, we assume that motion compensation for deviations from the flight path can be done perfectly via other algorithms.
- Finally, we assume that radar transmits bandpass linear-FM, or chirp, pulses $\mathbf{Re}\{s(t)\}$, where

$$s(t) = e^{j(\omega_0 t + \alpha t^2)} \mathrm{rect}\left(\frac{t}{T_p}\right) \tag{8.2}$$

  and

$$\mathrm{rect}(t) = \begin{cases} 1, & |t| \leq \frac{1}{2} \\ 0, & \text{otherwise} \end{cases} \tag{8.3}$$

  For convenience, we also define the baseband chirp pulse:

$$\tilde{s}(t) = e^{j\alpha t^2} \mathrm{rect}\left(\frac{t}{T_p}\right) \tag{8.4}$$

  Since the bandwidth of the pulse is approximately $2\alpha T_p$, a chirp pulse with a large time-bandwidth product can be designed through the selection of α and T_p. As we will show later,

a large bandwidth in the transmitted pulse results in high range resolution, while a long pulse is necessary to achieve high total pulse energy with limited peak transmission power. High pulse energy improves the SNR and therefore the detection rate at the receiver.

8.2 Range Resolution and Pulse Compression

Range resolution techniques are based on the concept of echo-ranging, which states that knowledge of an echo signal's round trip delay and its speed of propagation allows us to deduce the distance, or range, from the signal source to the target [20]. In a 2-D imaging situation, these techniques do not distinguish among targets which are equi-distant from the antenna, since their echoes all arrive at the same time. Instead, applying pulse compression on an echo due to a 2-D scene results in an estimate of the *integration* of reflectivity values from targets lying along constant range contours in the scene. Furthermore, these constant range contours are circular arcs, since the radar transmits spherical wavefronts. To simplify our discussion in this section, we assume without loss of generality that the 2-D scene is first "compressed" along the constant range contours into a line of scatterers. The range compression techniques presented here seek to recover a 1-D reflectivity function $q(y)$ from the echoed waveform. Furthermore, we assume a coordinate system centered at the middle of the radar footprint, with the footprint function

$$w_a(y) = \mathrm{rect}(y/W),$$

where W is the width of the footprint. The origin of the footprint is at a distance R from the radar, which corresponds to a wavefront travel time of $\tau_0 = 2R/c$.

Since the echo of a radar system consists of a superposition of echoes due to reflectors on the scene, it is mathematically expressed as the convolution between the scene reflectivity and the transmitted pulse. If the radar sends a short pulse of the form $\mathbf{Re}\{s(t)\}$, the echoed return due to a line of scatterers $q(y)$ is [20]

$$r(t) = \int \mathbf{Re}\{q(y)w_r(y)s(t - \tau_0 - \frac{2y}{c})\}dy. \tag{8.5}$$

In this case, a matched filter provides an optimal reconstruction of the reflectivity function $q(y)$. In this section, we derive two commonly used techniques for range resolution. The first involves a matched filter, while the second takes advantage of the form of the transmitted pulse to restructure the computation with an FFT.

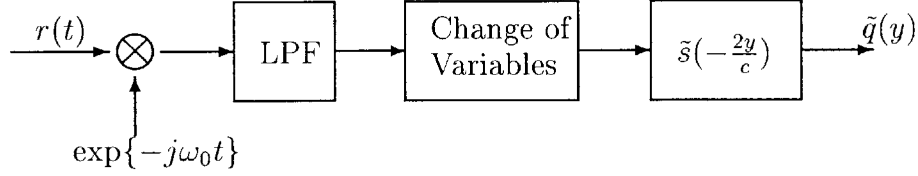


Figure 8-2: Range compression by matched filtering

8.2.1 Matched Filtering

A block diagram of the matched filtering approach to range resolution is shown in Figure 8-2. Essentially, the signal $r(t)$ first undergoes quadrature demodulation, and is then matched filtered to recover an estimate of the reflectivity function.

Quadrature demodulation consists of multiplying the signal with $\exp\{-j\omega_0 t\}$ and then lowpass filtering. The result is

$$r_1(t) = \int q(y)w_r(y)\tilde{s}(t-\tau_0-\frac{2y}{c})dy \tag{8.6}$$

$$= r_2(\frac{c(t-\tau_0)}{2}), \tag{8.7}$$

where

$$r_2(y) = [q(y)w_r(y)] * \tilde{s}(\frac{2y}{c}) \tag{8.8}$$

We filter $r_2(y)$ directly after first making the substitution $y = c(t-\tau_0)/2$. The matched filter has impulse response $\tilde{s}(-\frac{2y}{c})$, resulting in the following:

$$\tilde{q}(y) = [q(y)w_r(y)] * \tilde{s}(\frac{2y}{c}) * \tilde{s}(-\frac{2y}{c}) = [q(y)w_r(y)] * R_{\tilde{s}\tilde{s}}(\frac{2y}{c}), \tag{8.9}$$

where $R_{\tilde{s}\tilde{s}}(\cdot)$ is the autocorrelation of the baseband chirp signal. For chirps with large time bandwidth products,

$$R_{\tilde{s}\tilde{s}}(\frac{2y}{c}) \approx \frac{\sin(2\pi\alpha T_p y/c)}{2\pi\alpha T_p y/c} = \mathrm{sinc}(2\alpha T_p y/c)$$

Hence, using the matched-filtering method, $\tilde{q}(y)$ provides an estimate of $q(y)$ within the antenna footprint with a resolution of

$$c/(\alpha T_p).$$

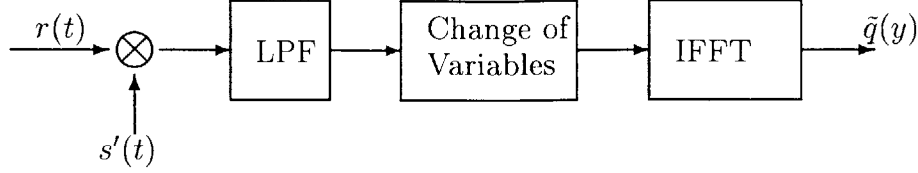


Figure 8-3: Range compression by deramp FFT; $s'(t) = \exp\{-j(\omega_0(t-\tau_0) + \alpha(t-\tau_0)^2)\}$

8.2.2 Deramp FFT

In the case that the footprint width $W \ll cT_p$, a simpler demodulation procedure called deramp-FFT can be used for range resolution. A block diagram of the process is shown in Figure 8-3.

To understand the structure of this algorithm, we first note that the return signal $r(t)$ can be expressed as

$$r(t) = \int \mathbf{Re}\left\{q(y)w_r(y)\exp\left\{j(\omega_0(t-\tau_0-\frac{2y}{c}) + \alpha(t-\tau_0-\frac{2y}{c})^2)\right\}\right\}dy. \tag{8.10}$$

over the time interval $t - \tau_0 \in [W/c - T_p/2, -W/c + T_p/2]$. (8.10) differs from (8.5) only in that $s(t)$ in (8.5) has finite extent, and hence (8.10) holds only for the stated region of t.

The first step involves multiplying $r(t)$ by

$$s'(t) = \exp\{-j(\omega_0(t-\tau_0) + \alpha(t-\tau_0)^2)\}$$

and then lowpass filtering. After some algebra, we derive the expression as

$$r_1(t) = \int q(y)w_r(y)\exp\left\{-j\frac{2y}{c}(\omega_0 + 2\alpha(t-\tau_0))\right\}\exp\left\{j\alpha\frac{y^2}{c^2}\right\}dy \tag{8.11}$$

For a sufficiently small footprint, it is often assumed that the quadratic phase term above is approximately 0, resulting in the following expression:

$$r_1(t) \approx \int q(y)w_r(y)\exp\left\{-jy\frac{2(\omega_0 + 2\alpha(t-\tau_0))}{c}\right\}dy = \tilde{Q}(u), \tag{8.12}$$

where $\tilde{Q}(\cdot)$ is the Fourier transform of $q(y)w_r(y)$, and $u = (2/c)(\omega_0 + 2\alpha(t-\tau_0))$. Hence under the given restrictions on the size of footprint and taking into account the restriction on the interval of t, $r_1(t)$ gives an estimate of the Fourier transform of $q(y)w_r(y)$ over the frequency range

$$u \in [\frac{2}{c}(\omega_0 + 2\alpha(\frac{W}{c} - \frac{T_p}{2})), \frac{2}{c}(\omega_0 - 2\alpha(\frac{W}{c} - \frac{T_p}{2}))].$$

With the assumption that $W \ll cT_p$,the frequency range can be expressed as

$$u \in \frac{2}{c}[\omega_0 - \alpha T_p, \omega_0 + \alpha T_p], \tag{8.13}$$

proportionate to the frequency band occupied by the chirp pulse. Inverse Fourier transforming results in an estimate of $q(y)$ with a resolution of $c/(\alpha T_p)$, as in the case of the matched filtering algorithm.

Even though the deramp-FFT algorithm is more efficient than the matched filtering algorithm, it has a few limitations. First, it assumes an exact knowledge of R, the distance from the antenna to the center of the radar footprint. An inexact estimate would result in phase errors which lead to blurring. Secondly, the size of the scene must be sufficiently small so that the quadratic phase term can be discarded. Finally, it can be used only when $W \ll cT_p$, which may be difficult to satisfy when the radar footprint is chosen to be large, as is often the case in stripmap SAR systems [20].

8.3 Stripmap Mode SAR

This section presents a unified framework for the analysis and derivation of stripmap mode SAR. We begin by describing the system geometry and our notation in Section 8.3.1. Section 8.3.2 derives the response of the system to a single isolated reflector at a known position. This is known as a point target response (PTR) and characterizes the entire imaging situation due to the linearity property discussed earlier. Then, the PTR is used in the derivations of three different image reconstructions algorithms in Sections 8.3.3–8.3.5. The range-Doppler algorithm, the ω-k algorithm, and the chirp-scaling algorithm are analyzed and their limitations discussed.

8.3.1 Imaging Geometry and Notation

The basic geometry for a side-looking strip-mapping SAR system is shown in Figure 8-4. The airborne radar platform moves in a straight line along the azimuth direction, while the antenna points at the ground at a fixed angle of elevation and normal to the flight path. Pulses are transmitted and received at regular intervals along a section of the flight path called the *synthetic aperture*. The radar is mounted on a plane for *airborne* SAR, and on a satellite for *spaceborne* SAR. The radar footprint of airborne SAR systems is generally smaller because of the lower elevation of planes. In this section, the direction at which the radar points is assumed to be perpendicular to the flight path at all times.

In our discussions, the coordinates for the 2-D reflectivity function (and for the reconstructed image) are (x, y), where x corresponds to the azimuth direction, and y to the slant-range. The corresponding coordinates in the frequency domain are ω_x and ω_y, where ω_x is often also called the Doppler frequency. In the data space, the collected echoes are described as functions of (x, t),

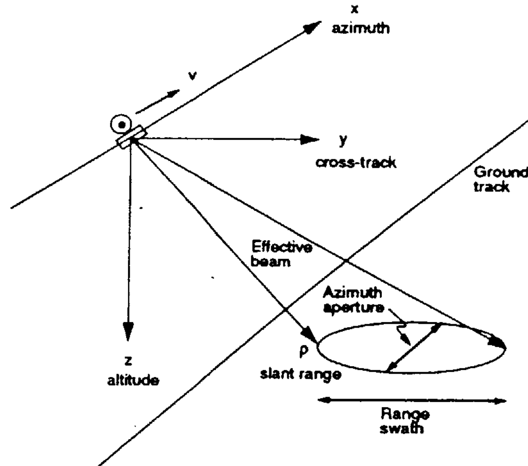


Figure 8-4: Geometry for stripmap mode SAR. From [39].

where x corresponds to the position of the radar on the flight path, while t is the time dimension for each returned pulse. In the frequency domain, we use the coordinate system (ω_x, ω).

The area covered by the footprint is represented by a windowing function $w(x, y)$ often approximated as a separable function expressible by $w_a(x)w_r(y)$, where

$$w_a(x) = \mathrm{rect}(x/L) \tag{8.14}$$

$$w_r(y) = \mathrm{rect}((y - R)/W) \tag{8.15}$$

Hence the footprint size is W by L, and its center is at $(0, R)$, where R as the slant range distance between the antenna and the center of the footprint.

8.3.2 Point Target Response

Consider a scene consisting of a single point reflector:

$$q(x, y) = \delta(x, y - y_0).$$

The echo due to a chirp pulse $\mathbf{Re}\{s(t)\}$ sent when the antenna is at azimuth position x is

$$w_a(x)w_r(y_0)\mathbf{Re}\left\{s\left(t - \frac{2R(x, y_0)}{c}\right)\right\}, \tag{8.16}$$

where $R(x, y)$ represents the radial range between the point reflector and the antenna:

$$R(x, y) = \sqrt{x^2 + y^2} \tag{8.17}$$

After quadrature demodulation, the PTR can be shown to be

$$\begin{aligned} h(x,t|y_0) &= w_a(x)\exp\left\{-\frac{2j\omega_0}{c}R(x,y_0)\right\}\tilde{s}\left(t-\frac{2R(x,y_0)}{c}\right) && (8.18)\\ &= h_1(x,t|y_0) * h_2(x,t) && (8.19)\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h_1(x-x',t-t'|y_0)h_2(x',t')dx'dt' && (8.20)\end{aligned}$$

where $\tilde{s}(\cdot)$ is as defined in (8.4), and $h_1(\cdot)$ and $h_2(\cdot)$ are defined as follows. The dispersion in the range direction is described by

$$h_2(x,t) = \delta(x)\tilde{s}(t). \tag{8.21}$$

It is clearly spatially invariant in both range and azimuth directions.

The dispersion in the azimuth direction is described by

$$h_1(x,t|y_0) = w_a(x)\exp\left\{-\frac{2j\omega_0}{c}R(x,y_0)\right\}\delta(t-2R(x,y_0)/c). \tag{8.22}$$

It is a 2-D range-variant function whose locus of energy is traced out by the argument of the Dirac delta function. An examination of the form of (8.17) shows that the locus is hyperbolic, with its curvature parameterized by the range y_0. This phenomenon in which the range variation of $h_1(x,t|y_0)$ spans more than a single range resolution cell is called *range migration*. The linear term of the range migration is called the *range walk*, while the higher ordered terms are called the *range curvature*. Because the size of the radar footprint is considerably smaller in airborne SAR, range migration can be considered negligible in airborne SAR. As a result, processing algorithms for airborne SAR are considerably simpler, and our discussions in this chapter focus on spaceborne SAR instead.

The complete PTR $h(x,t|y_0)$ is both two-dimensional and range-variant. The difficulty with designing efficient SAR processing algorithms is primarily due to these two key characteristics. If the PTR is approximately 1-D due to insignificant range migration, then only a sequence of range-variant 1-D cross-correlations needs to be applied. If, on the other hand, the range variation is very small, a 2-D space-invariant filter can be used. With a 2-D range-variant PTR, the direct approach of cross-correlation in the azimuth-time domain is highly computationally inefficient, and alternative methodologies are needed.

In the following derivations and analyses of the algorithms, we will find it useful to compute the Fourier transforms of $h(x,t|y_0)$. In particular, it can be shown using the principle of stationary

phase that [35]

$$h^t(x,\omega|y_0) = C_1 a\left(x, \frac{\omega}{2\alpha}\right) \exp\left\{-j\left(\frac{2\omega_0}{c}+\omega\right) R(x,y_0)\right\} \exp\left\{-j\frac{\omega^2}{4\alpha}\right\} \quad (8.23)$$

$$H(\omega_x,\omega|y_0) = C_2 a\left(-\frac{cy_0\omega_x}{2\omega_0}, \frac{\omega}{2\alpha}\right) \exp\left\{-jy_0\left[\left(\frac{\omega+\omega_0}{c/2}\right)^2 - \omega_x^2\right]^{1/2} - j\frac{\omega^2}{4\alpha}\right\} \quad (8.24)$$

$$h^x(\omega_x,t|y_0) = C_3 a\left(-\frac{cy_0}{2\omega_0}\omega_x, t-\frac{2y_0}{c}R_\omega(\omega_x)\right) \exp\left\{j\alpha'\left[t-\frac{2y_0}{c}R_\omega(\omega_x)\right]^2\right\} \cdot \exp\left\{-j\frac{2\omega_0}{c}y_0\sqrt{1-\frac{c^2\omega_x^2}{4\omega_0^2}}\right\}; \quad (8.25)$$

where

$$a(x,t) = w_a(x)\mathrm{rect}(t/T_p) \quad (8.26)$$

$$\alpha'(\omega_x,y_0) = \alpha\left\{1+\frac{2\alpha y_0\omega_x^2}{\left(4\omega_0^2/c^2-\omega_x^2\right)^{3/2}}\right\}^{-1} \quad (8.27)$$

$$R_\omega(\omega_x) = \left\{1-\frac{\omega_x^2}{4\omega_0^2/c^2}\right\}^{-1/2}. \quad (8.28)$$

C_1, C_2, and C_3 are complex constants, which will be dropped in the following presentations for convenience.

For a general scene reflectivity function $q(x,y)$, the collected data is represented by

$$g(x,t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} q(x',y')w_r(y')h(x-x',t|y')dx'dy' \quad (8.29)$$

Even though our notation indicates a coordinate system of continuous variables, it is important to remember that $g(x,t)$ is sampled on a rectangular grid. In the azimuth direction, the sampling rate is precisely the pulse repetition frequency (PRF) of the radar along the flight path. In the range direction, the sampling rate refers to the rate at which the echoes are sampled and recorded over time.

8.3.3 Range-Doppler Algorithm

As can be seen in (8.29), the data collection process represents a 2-D filtering operation on the scene reflectivity function, where the filter is range-varying and can be decomposed into the form of (8.19). Hence, one strategy for recovering $q(x,y)$ is to apply a matched filter approximating $h^*(-x,-t|y_0)$ to the data. The range-Doppler algorithm (RDA) applies precisely this strategy. It is the classical algorithm for stripmap SAR, and there exist many variations on its implementation.

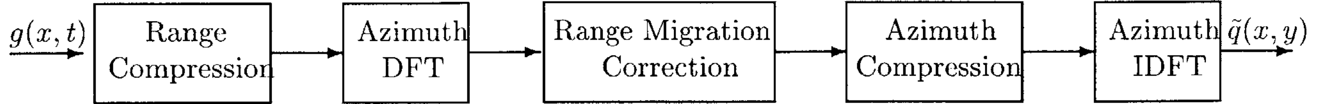


Figure 8-5: Flow diagram for the range-Doppler algorithm.

Our description here is based on [3], which was applied to spaceborne data with significant range migration.

Because $h(x,t|y_0)$ is decomposable into the form of (8.19), RDA performs the matched filtering first along the range direction, by applying the filter $h_2(x,-t)$ for each x. This corresponds to applying range compression as described in Section 8.2.1. Then, the rest of the algorithm seeks to apply the 2-D range-varying filter $h_1^*(-x,-t|y)$ efficiently. This is done by first performing range migration correction to "straighten" the hyperbolic energy locus of (8.22), and then applying range-dependent 1-D filters in the azimuth direction. A detailed block diagram of the processing steps is shown in Figure 8-5. As implied by the name of the algorithm, it is most convenient to base our derivation in the (ω_x, t) plane, where azimuth compression is performed.

The first step of RDA consists of range compression on each of the returned pulses. Mathematically, the azimuth Fourier transform of its result can be expressed as a convolution between (8.25) and $\tilde{s}(-t)$:

$$\begin{aligned} f(\omega_x, t) = \; & C_3 a\left(-\frac{c y_0}{2\omega_0}\omega_x, t - \frac{2y_0}{c}R_\omega(\omega_x)\right) \\ & \cdot \mathrm{sinc}\left(\alpha T_p(t - \frac{2y_0}{c}R_\omega(\omega_x))\right) \\ & \cdot \exp\left\{-j\frac{2\omega_0}{c}y_0\sqrt{1-\frac{c^2\omega_x^2}{4\omega_0^2}}\right\}. \end{aligned} \tag{8.30}$$

Here we have made the first approximation of RDA, namely that $\alpha' = \alpha$. The approximation holds if ω_x is small as compared with $2\omega_0/c$; it breaks down when the imaging setup results in large ω_x, which happens with a large radar squint angle. In such cases, defocusing in the range direction occurs unless a secondary range migration procedure is applied [21].

We now consider the locus of points which maximizes the sinc($\cdot$) term in (8.30):

$$t(\omega_x, y_0) = \frac{2}{c}y_0\left\{1 - \frac{\omega_x^2}{4\omega_0^2/c^2}\right\}^{-1/2} \tag{8.31}$$

To correct or "straighten" this curve, we perform a ω_x-dependent time compression and shift.

Namely, for each ω_x, shift by

$$\Delta t(\omega_x, y_0) = \frac{2}{c} y_0 \left(1 - \left\{ 1 - \frac{\omega_x^2}{4\omega_0^2/c^2} \right\}^{-1/2} \right),$$

so that the migration curve becomes $t = \frac{2}{c} y_0$, independent of ω_x. This operation requires 1-D interpolations along the t-axis, which can cause artifacts in the image if not done with sufficiently high precision.

It now remains to perform azimuth compression via multiplication by

$$\exp \left\{ j \frac{2\omega_0}{c} y_0 \sqrt{1 - \frac{c^2 \omega_x^2}{4\omega_0^2}} \right\}$$

along each range row y_0. Taking the inverse Fourier transform along the azimuth direction, and substituting $t = 2y/c$ results in

$$\tilde{q}(x, y) = \operatorname{sinc} \left(\frac{\omega_0 L}{c y_0} x \right) \operatorname{sinc} \left(\alpha T_p (y - y_0) \right)) \tag{8.32}$$

The azimuth resolution can be seen to be

$$c y_0 / (2\omega_0 L) = D/2, \tag{8.33}$$

where D is the diameter of antenna aperture or lens. The azimuth resolution *improves* with a smaller antenna size in SAR and is independent of the range. This is in contrast to the expression for the azimuth resolution of a conventional radar system (8.1), for which resolution worsens with smaller antennas.

8.3.4 ω-k Migration Algorithm

The ω-k migration algorithm, first used in the geophysics field for seismic surveying, corrects range migration exactly. Its name comes from the fact that range migration is corrected in the 2-D Fourier transform domain; in the literature, ω and k refer to the frequency domain correspondents of t and x, respectively. In order to be consistent with our earlier discussions, however, we use (ω_x, ω) to denote the frequency domain correspondents of (x, t).

The original derivation of this algorithm is based on the wave equation and the concept of "back-propagation" of the wave-field measurements at the sensors [8]. However, here we derive the algorithm using Fourier transform identities and the principle of stationary phase[1]. To aid in our description, a flow diagram of the algorithm in shown in Figure 8-6.

Our derivation starts with (8.24), the PTR in the (ω_x, ω) plane. Range compression effectively

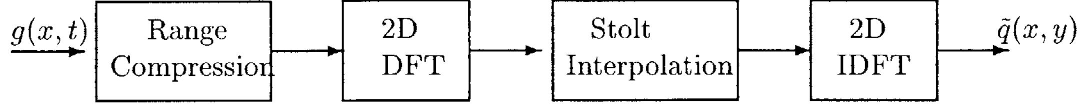


Figure 8-6: Flow diagram of the ω-k migration algorithm

cancels the ω^2 phase term in (8.24), resulting in a ranged-compressed PTR of

$$F(\omega_x,\omega) = a(-\frac{cy_0\omega_x}{2\omega_0},\frac{\omega}{2\alpha})\exp\left\{-jy_0\left[\left(\frac{\omega+\omega_0}{c/2}\right)^2-\omega_x^2\right]^{1/2}\right\} \tag{8.34}$$

At this point, we make the following substitution of variables:

$$\omega_y = \left[\left(\frac{\omega+\omega_0}{c/2}\right)^2-\omega_x^2\right]^{1/2}, \tag{8.35}$$

and re-write (8.34) as

$$F(\omega_x,\omega_y) = a(-\frac{cy_0\omega_x}{2\omega_0},\frac{1}{2\alpha}\left(\frac{c}{2}\sqrt{\omega_y^2+\omega_x^2}-\omega_0\right))\exp\{-jy_0\omega_y\} \tag{8.36}$$

It is now clear that the inverse 2-D Fourier transform of the above approximates the scene reflectivity function $\delta(x,y-y_0)$. We have derived the ω-k migration algorithm using only the Fourier transform and the principle of stationary phase. Notice the only assumption needed in its derivation is a large space-bandwidth product in the signal in the azimuth direction (used in an application of the principle of stationary phase). Since that assumption is easily satisfied in SAR systems, the ω-k algorithm represents an exact reconstruction.

The only drawback to the algorithm is in the implementation of the change of variables in (8.35). This step, called the Stolt interpolation, requires precise implementation to avoid aliasing artifacts in the final image. In comparison, the 1-D interpolators used the RDA can tolerate shorter interpolation kernels, which are chosen for sake of efficiency.

8.3.5 Chirp-Scaling Algorithm

The range-Doppler algorithm and the ω-k migration algorithm both require the use of interpolators. In this section we present the chirp-scaling algorithm(CSA), which uses only 1-D Fourier transforms and multiplications in its implementation. The CSA consists of 7 distinct steps, as shown in Figure 8-7. It is more accurate than the range-Doppler algorithm, and its only limitation is that the transmitted waveforms must be chirp pulses.

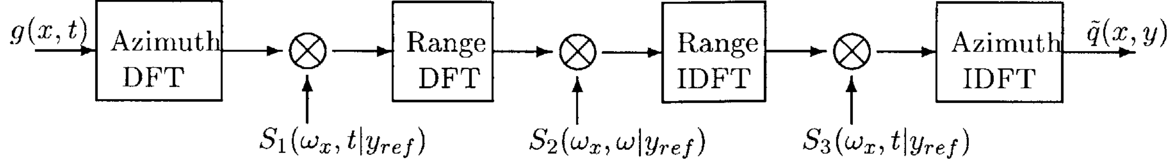


Figure 8-7: Steps of the chirp-scaling algorithm.

The chirp-scaling algorithm begins by Fourier transforming the data along the x-axis, where the point target response is expressed by (8.25). To simplify the following analysis, we will drop the windowing term $a(\cdot)$ in (8.25), so that

$$h(\omega_x, t|y_0) = \exp\left\{j\alpha'\left[t - \frac{2y_0}{c}R_\omega(\omega_x)\right]^2\right\}\exp\left\{-j\frac{2\omega_0}{c}y_0\sqrt{1-\frac{c^2\omega_x^2}{4\omega_0^2}}\right\}. \qquad (8.37)$$

The first step is to "equalize" all of the range migration trajectories to match a reference trajectory

$$t_{ref}(\omega_x) = \frac{2}{c}y_{ref}R_\omega(\omega_x)$$

To do that, we multiply $h(\omega_x, t|y_0)$ by

$$S_1(\omega_x, t|y_{ref}) = \exp\{-j\alpha'(R_\omega(\omega_x) - 1)(t - t_{ref}(\omega_x))^2\}. \qquad (8.38)$$

After some algebra, the result can be shown to be

$$\begin{aligned} P(\omega_x, t|y_0, y_{ref}) &= h(\omega_x, t|y_0)S_1(\omega_x, t|y_{ref}) \\ &= \exp\left\{j\hat{\alpha}\left(t - \frac{2}{c}((y_0 - y_{ref}) + y_{ref}R_\omega(\omega_x))\right)^2\right\} \\ &\quad \cdot\exp\left\{j\hat{\alpha}(R_\omega(\omega_x) - 1)\left(\frac{2}{c}(y_0 - y_{ref})\right)^2\right\} \\ &\quad \cdot\exp\left\{-j\frac{2\omega_0}{c}y_0\sqrt{1-\frac{c^2\omega_x^2}{4\omega_0^2}}\right\}, \end{aligned} \qquad (8.39)$$

where

$$\hat{\alpha}(\omega_x, y_0) = \alpha'(\omega_x, y_0)R_\omega(\omega_x). \qquad (8.40)$$

The first phase term of (8.39) reflects the primary result of this "chirp-scaling" step. The phase

minimum of every range chirp has moved to

$$t = \frac{2}{c}(y_0 - y_{ref}) + \frac{2}{c} y_{ref} R_\omega(\omega_x);$$

The curvature of the range migration curve now depends on y_{ref} rather than y_0. As a result, all of the range migration curves have the same curvature. The side effects of this operation are the change in the chirp rate from α' to $\hat{\alpha}$, and the added second phase term of (8.39). The chirp rate difference must be taken into consideration during range compression, while the second phase term can be compensated after range compression.

We now take the range Fourier transform of (8.39) and perform partial range and azimuth compression using a filter adjusted to the new chirp rate and fixed to the reference range y_{ref}. Referring to (8.24), we can derive the following expression for the filter:

$$S_2(\omega_x, \omega | y_{ref}) = \exp\left\{ -j y_{ref} \left[\left(\frac{\omega + \omega_0}{c/2} \right)^2 - \omega_x^2 \right]^{1/2} + j \frac{\omega^2}{4\hat{\alpha}(\omega_x, y_{ref})} \right\}. \tag{8.41}$$

The filter performs range compression with respect to the first phase term of (8.39), and performs perfect azimuth compression only for the reference range. Notice that we have assumed that the variation of $\hat{\alpha}(\cdot)$ due to range variations is small enough that

$$\hat{\alpha}(\omega_x, y_0) \approx \hat{\alpha}(\omega_x, y_{ref})$$

After this step, the range migration is effectively eliminated.

Next, the range inverse Fourier transform is applied to the data, and the remaining phase terms of 8.39 are canceled out by multiplying with

$$\begin{aligned} S_3(\omega_x, t | y_{ref}) \quad &= \quad \exp\left\{ -j\hat{\alpha}(R_\omega(\omega_x) - 1) \left(t - \frac{2y_{ref}}{c} \right)^2 \right\} \\ &\quad \cdot \exp\left\{ j \frac{2\omega_0}{c} \left(\frac{ct}{2} - y_{ref} \right) \sqrt{1 - \frac{c^2 \omega_x^2}{4\omega_0^2}} \right\}. \end{aligned} \tag{8.42}$$

Notice that the substitution $t = 2y_0/c$ can be made here because range migration has already been corrected. At the end of the multiplication, the phase terms have all been effectively canceled, so that the focussed image can be obtained by a final inverse azimuth Fourier transform.

8.4 Functional Equivalence Among SAR Algorithms

The previous section presents three very different image formation algorithms for stripmap SAR. The RDA is an approximate algorithm whose most notable implementation requirement is a sequence of 1-dimensional interpolations onto a uniform grid. The ω-k algorithm is exact, but requires 2-dimensional interpolation in the frequency domain. The chirp-scaling algorithm requires only 1-dimensional Fourier Transforms and multiplications.

However, RDA, ω-k, CSA can be best viewed as *classes* of algorithms, because many of the individual steps can be implemented in many different ways. The following is representative list of possible implementations of some of major steps in the algorithms:

- FIR Filtering:
  - direct convolution in time
  - multiplication in frequency domain
- 1-dimensional Fourier transforms and inverse Fourier transforms:
  - Direct evaluation
  - FFT and IFFT
  - the Goertzel algorithm
  - the chirp transform
  - Winograd Fourier Transform
- Bandlimited 1-dimensional interpolation on a regular grid:
  - Direct evaluation by upsampling, lowpass filtering, and then downsampling
  - Staged implementations
  - A variety of different polyphase/multirate implementations
- 2-dimensional Fourier transforms and inverse Fourier transforms:
  - Direct evaluation
  - 2-dimensional FFT and inverse FFT
  - Decomposition into 2 groups of 1-dimensional Fourier transforms, which in turn can be implemented in a variety of ways.
- Stolt interpolation[25]:
  - Sinc interpolation for each desired output point, via a sequence of inner products

- Oversample the grid, and then choose the near neighboring point; oversampling requires 2-D uniform interpolation, performed by sequences of bandlimited 1-dimensional interpolation

Finally, the functional equivalence of these algorithms depends on the conditions of the data collection process. If the transmitted pulses are not chirp pulses, then the deramp-FFT algorithm cannot be used for the range compression part of both RDA and the ω-k algorithms, and the CSA is not applicable. On the other hand, if the range migration problem is severe, the RDA algorithm may not result in satisfactory performance.

8.5 Summary

Image formation from synthetic aperture radar data represents an important, computationally intensive DSP application which would greatly benefit from the efficient use of resources in a networked computing environment. Moreover, as this chapter has shown, the algorithms for accomplishing this task are naturally expressible by a composition of DSP primitive functions, and the implementational alternatives are rich and interesting. Both of these are prime reasons supporting our choice to use it as the processing task in the simulation study of Chapter 9.

Chapter 9

Simulation with SAR Algorithms

In this chapter, we present the results of one set of experiments performed on the simulation environment on the application of image formation for stripmap mode synthetic aperture radar. Section 9.1 begins the chapter by describing the experimental setup in the simulation. Section 9.2 examines the system behavior under packet-wise cost minimization and system-wide optimization, and compares the packet delay achieved with the two strategies. The results indicate that the system optimization strategy resulted in significantly lower packet delay than the packet-wise cost minimization strategy. Section 9.3 then presents results on the adaptation of the execution path selection processes to changes in the computing environment.

We note here that the setup of the experimental environment necessarily involves a large number of parameters, only a subset of which is carefully selected for study, with the other choices being somewhat arbitrary. Therefore, the study presented in this chapter is intended to be only a rough assessment of the relative performance and implementational complexity of the algorithm adaptation strategies, rather than a definitive evaluation.

9.1 Experimental Setup

In total, there are 50 processors and 1 sensor in the simulated environment. Figure 9-1 is a table of the processor rates associated with each primitive functionality in the simulated environment. For example, there are 3 virtual processors implementing the sequence multiplication function; their processing rates are 300, 1000, and 1500 data units/s, respectively. Relatively speaking, the rates of the processors implementing up- and down-sampling are the highest, and the rates of processors implementing 2-dimensional functions (e.g., 2-D DFT) are lower than those implementing 1-dimensional functions (e.g., 1-D DFT). Primitive functionalities to disassemble and assemble data blocks are not listed, but they are abundantly available in the simulation environment.

Each data block produced by the sensor represents an entire set of collected echoes, which is converted to an actual image as a result of the processing. During the simulations, the production rate of each sensor is varied to observe the system performance at various operating points. The tagged algebraic expression incorporates all of the implementational variations described in

Functionality	Processor rates	Functionality	Processor rates
Sequence Multiplication	{300, 1000, 1500}	Low-pass filter	{200, 300, 800}
Convolution	{200, 300, 400 }	1-D interpolation	{300,300, 500}
1-D DFT	{400, 200}	1-D FFT	{300, 300, 500}
DFT via Goertzel	{200, 300}	DFT via Chirp Transform	{200, 400}
1-D inverse DFT	{200, 300}	1-D inverse FFT	{300, 400, 600}
Upsampling	{4000, 4000, 4000}	Downsampling	{4000, 4000}
2-D FFT	{100, 200, 300}	2-D DFT	{100, 100}
2-D IFFT	{50, 100, 150}	2-D IDFT	{100, 200}
Stolt interpolation	{200, 300}	Inner product	{100, 200, 500, 500}

Figure 9-1: Computation environment: the processor rates for virtual processors implementing each primitive functionality.

Section 8.4.

9.2 System Behavior

With the simulation environment setup, we are now ready to examine the system behavior under the different schemes of algorithm adaptation. We begin by looking at how the different algorithm adaptation strategies assign execution paths to data blocks as the production rate of the sensor varies.

The top graph of Figure 9-2 shows how the execution path allocations vary as a function of the production rate of the sensor when packet-wise cost minimization is used. Each line on the graph corresponds to the average rate or throughput at which the labeled execution path outputs data blocks. Notice that all three of the major image formation algorithms, CSA, RDA, and ω-k, are used for all production rates. The bottom graph shows the number of active execution paths observed over a 2 (simulation) minute interval as a function of the production rate λ. Note that a large number of execution paths can be simultaneously active.

In comparison, the algorithm path allocations resulting from system-wide optimization is plotted in the top graph of Figure 9-3. The bottom graph shows the number of active execution paths observed over a 2 (simulation) minute interval as a function of the production rate λ. Unlike the allocations obtained by packet-wise minimization, the ω-k algorithm was not used for production rates below 800 units/s. Notice also that the number of observed active execution paths is significantly lower than that when packet-wise minimization is used. This is largely a result of the fact that the packet-wise minimization strategy makes choices as it reaches each decision node during the processing of the data, while the system-wise optimization strategy makes the decision at the initialization of the data block.

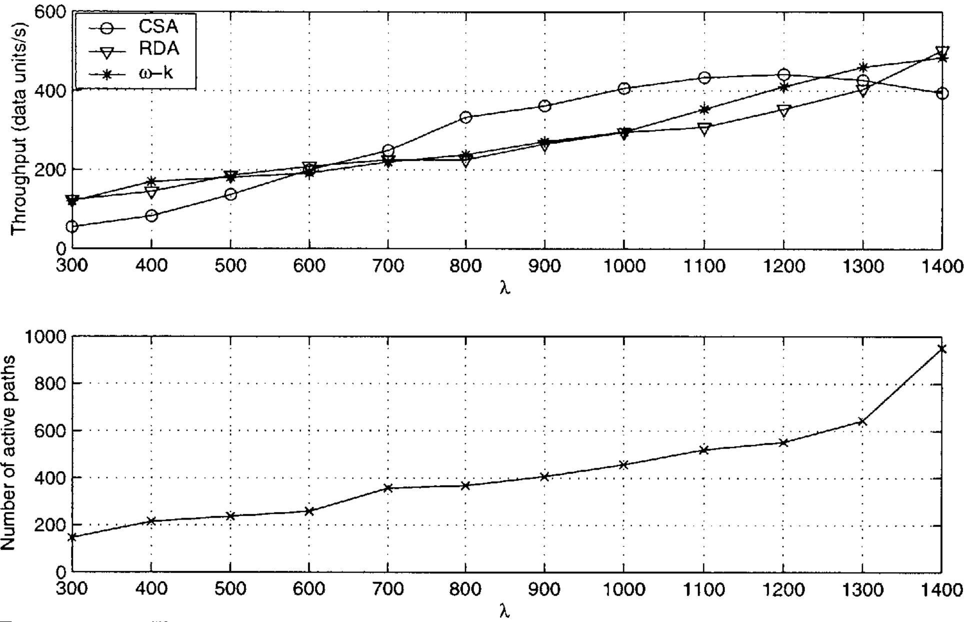


Figure 9-2: Execution path allocations using packet-wise cost minimization

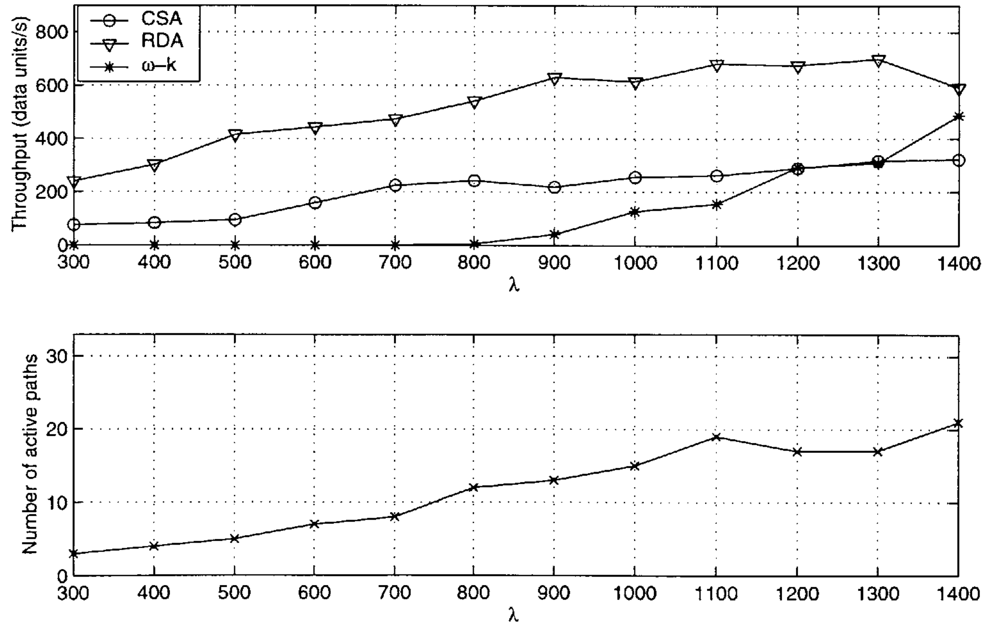


Figure 9-3: Execution path allocations using system optimization

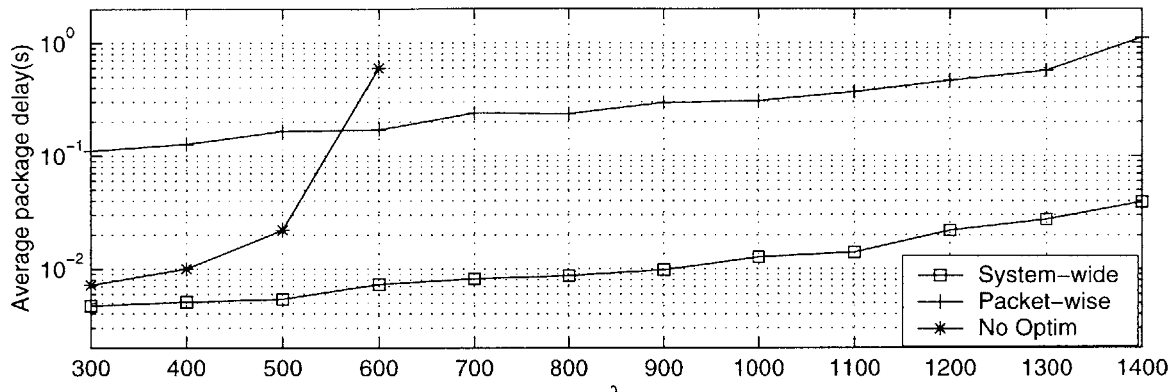


Figure 9-4: Packet delay achieved by using algorithm adaptation via system-wide optimization, algorithm adaptation via packet-wise optimization, and no adaptation.

Figure 9-4 compares the average packet delay achieved by the different algorithm adaptation strategies and with no algorithm adaptation, under varying values of the sensor production rate. As we can see, the system-wide optimization strategy achieves a significantly lower packet delay than packet-wise cost minimization. However, both algorithm adaptation strategies resulted in bounded packet delay over a wide range of sensor production rates, whereas using no algorithm adaptation resulted in a significant degradation in packet delay even under low sensor production rates.

9.3 Reaction to Sudden Changes

The previous section examined the system behavior with algorithm adaptation under steady state conditions. In this section, we look at how the execution path selections adapt to three different types of changes in the computation environment.

9.3.1 Reaction to Processor Failure

In the first experiment, we study the system behavior when processors fail. The scenario is that 4 processors implementing the 1-D Fourier Transform functions fail simultaneously, so that after the failure, the system's processor distribution is as described in Figure 9-5. This set of processor failures effectively reduces the system's ability to use 1-dimensional Fourier Transforms by over 50%. For the experiment, the production rate λ is fixed at 900 units/s, and the failures occur at Time = 30.

Figure 9-6 shows how the execution path allocations and packet delay react to this change in the computation environment when the packet-wise optimization is used. Even though the curves are not steady, a careful examination of the top graph reveals that following the failure, the system usage of the ω-k algorithm increased, while the use of the other two algorithms decreased slightly.

Functionality	Processor rates	Functionality	Processor rates
Sequence Multiplication	{300, 1000, 1500}	Low-pass filter	{200, 300, 800}
Convolution	{200, 300, 400 }	1-D interpolation	{300, 300, 500}
1-D DFT	**{ 200}**	**1-D FFT**	**{300, 300}**
DFT via Goertzel	**{200}**	**DFT via Chirp Transform**	**{200}**
1-D inverse DFT	{200, 300}	1-D inverse FFT	{300, 400, 600}
Upsampling	{4000, 4000, 4000}	Downsampling	{4000, 4000}
2-D FFT	{100, 200, 300}	2-D DFT	{100, 100}
2-D IFFT	{50, 100, 150}	2-D IDFT	{100, 200}
Stolt interpolation	{200, 300}	Inner product	{100, 200, 500, 500}

Figure 9-5: Computation environment after processor failure. Entries which changed from Figure 9-1 are in **bold**.

The packet delay showed a corresponding rise following the processor failures.

Figure 9-7 shows the equivalent data when system-wide optimization is used. The average packet delay increases significantly immediately following processor failures, but then subsequently improves dramatically as the execution path allocations shift. A comparison of the execution path allocations before and after the failure shows that the system's use of CSA, which relies on the use of 1-D DFT functions, drops dramatically. The ω-k algorithm, which is more reliant on the 2-D DFT functions, becomes more heavily used instead.

A comparison of Figures 9-6 and 9-7 deserves comment in several areas. First, the behavior of packet-wise optimization in steady state is clearly more variable that that of system-wide optimization. This is in line with our understanding of the possibly unstable behavior of packet-wise optimization. Second, immediately following the processor failures, the average packet delay changes much more dramatically for system-wide optimization than for packet-wise optimization. This is due to the fact that the packet-wise optimization responds immediately to increased load. On the other hand, it takes several iterations of the gradient projection algorithm for system-wide optimization to fully react to the new environment.

9.3.2 Reaction to Functionality Change

The second experiment looks at the case in which the functionality of some processors change. The scenario is that two of the virtual processors implementing 1-dimensional Interpolation combines to form a single virtual processor implementing Stolt interpolation. The processor distributions following the failure is as shown in Figure 9-8. For the experiment, the production rate λ is fixed at 900 units/s, and the failures occur at Time = 30.

Plots showing how the execution path allocations and packet delay react to this change in the computation environment are shown in Figure 9-9 and 9-10. Figure 9-9 reflects the result when packet-wise minimization is used, while Figure 9-10 shows the corresponding results for system-wide

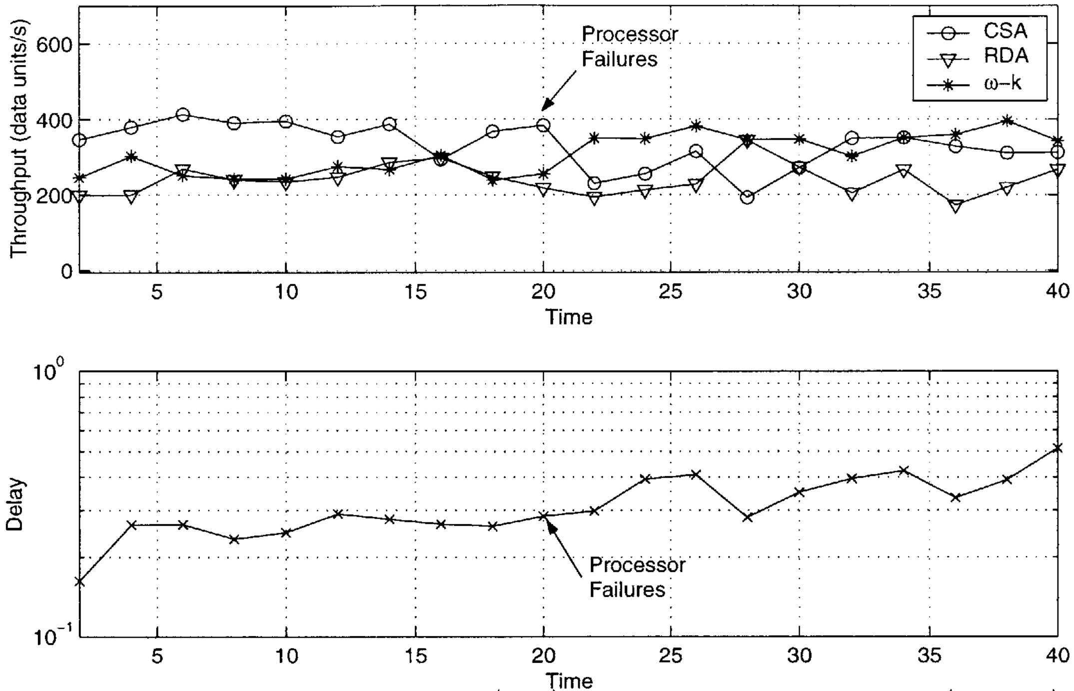


Figure 9-6: Under packet-wise adaptation, (Top) Execution path allocations (Bottom) Average packet delay when processors failures occur at Time = 30.

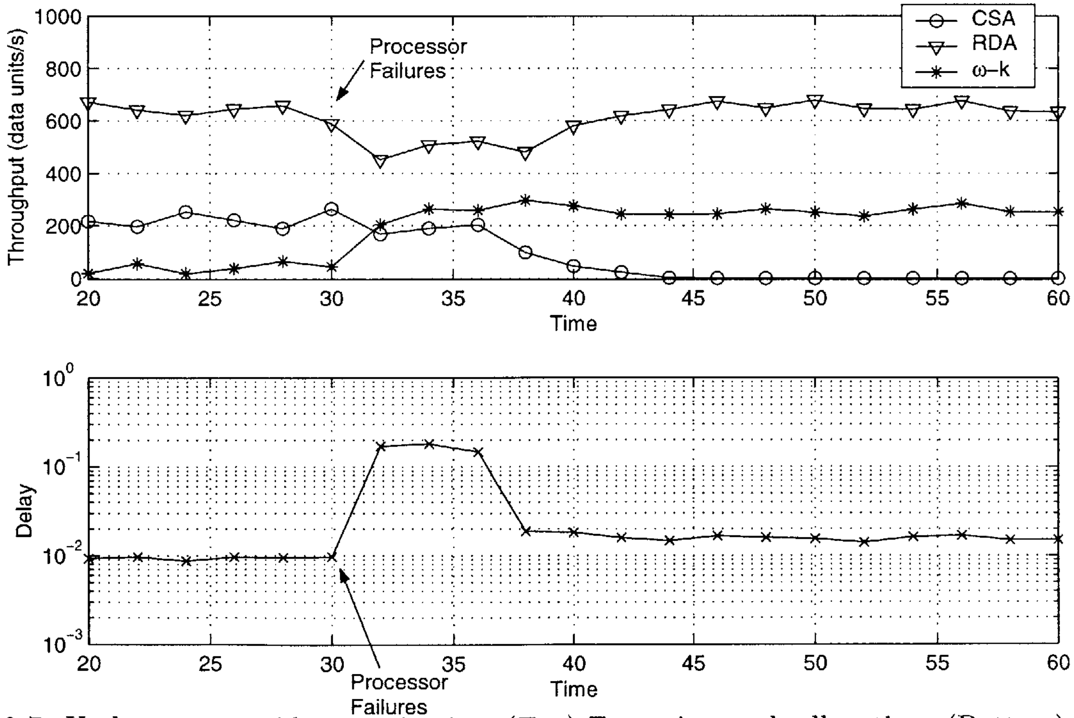


Figure 9-7: Under system-wide optimization, (Top) Execution path allocations (Bottom) Average packet delay when processors failures occur at Time = 30.

Functionality	Processor rates	Functionality	Processor rates
Sequence Multiplication	{300, 1000, 1500}	Low-pass filter	{200, 300, 800}
Convolution	{200, 300, 400}	**1-D interpolation**	**{300}**
1-D DFT	{400, 200}	1-D FFT	{300, 300, 500}
DFT via Goertzel	{200, 300}	DFT via Chirp Transform	{200, 400}
1-D inverse DFT	{200, 300}	1-D inverse FFT	{300, 400, 600}
Upsampling	{4000, 4000, 4000}	Downsampling	{4000, 4000}
2-D FFT	{100, 200, 300}	2-D DFT	{100, 100}
2-D IFFT	{50, 100, 150}	2-D IDFT	{100, 200}
Stolt interpolation	**{200, 300, 1000}**	Inner product	{100, 200, 500, 500}

Figure 9-8: Computation environment after processor changes

optimization.

The top graph of Figure 9-9 shows that following the processor change, the use of the ω-k algorithm increased dramattically, while the use of RDA decreased dramatically. This agrees well with our intuition because the 1-D interpolation function, used mostly for RDA, has become less available, while the Stolt interpolation function, used mostly for the ω-k algorithm, has become much more available. The bottom graph of Figure 9-9 shows that processor changes cause minimal changes in average packet delay.

The top graph of Figure 9-10 shows the changes in execution path selection when system-wide optimization is used. Like the case with the packet-wise optimization, there is a large drop in the usage of RDA following the processor changes. This occurs in conjunction with the rise of the use of both the CSA and the ω-k algorithms. Clearly the use of the ω-k algorithm rises because of the higher availability of the Stolt interpolation primitive. The CSA algorithm also sees increased usage because while it does not use the 1-D interpolation function, it shares the use of the 1-D Fourier Transform and multiplication primitives with the RDA algorithm. Some of those resources becomes freed when the use of the RDA algorithm is diminished, and they can be used to implement the CSA algorithm instead. The bottom graph of Figure 9-10 shows a significant transient effect in the average packet delay, which increases significantly immediately following processors change, but then subsequently improves dramatically as the execution path allocations gradually shift.

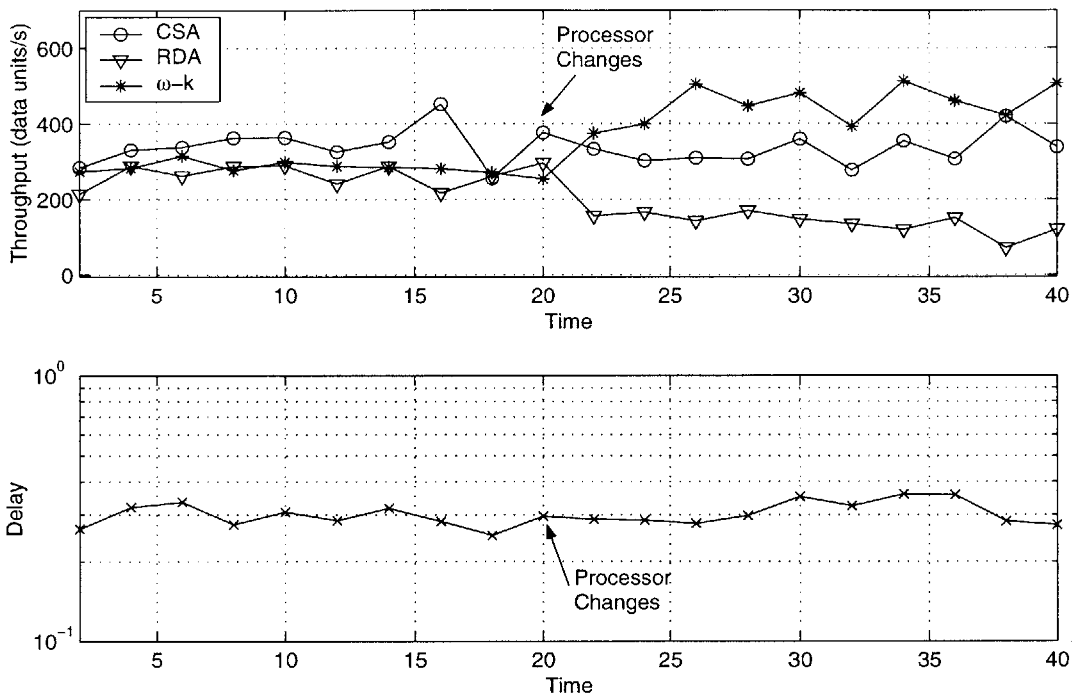


Figure 9-9: Packet-wise adaptation: Reaction to processor changes.

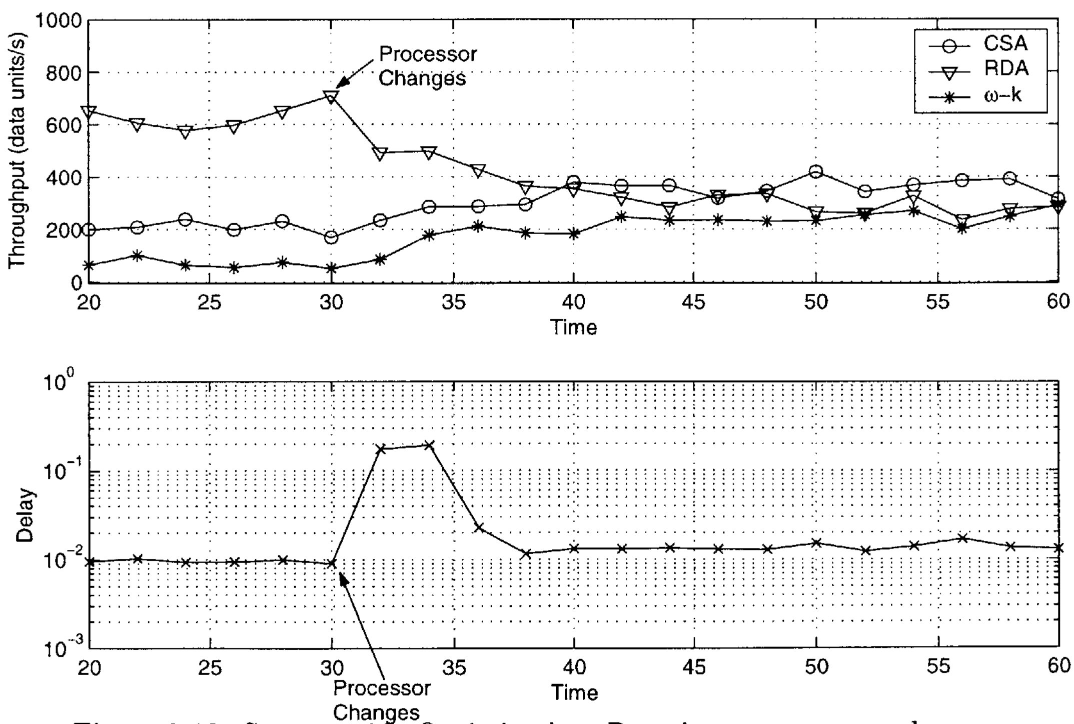


Figure 9-10: System-wide Optimization: Reaction to processor changes.

9.3.3 Reaction to Change in Workload

Finally, we examine a scenario in which the processor distribution does not change but the sensor production rate changes dramatically. In this experiment, the sensor production rate begins at 500 units/s, jumps to 900 units/s, and then drops back to 500 units/s.

Figure 9-11 shows the reaction of the packet-wise optimization strategy to these sudden changes. An increase in production rate draws a significantly higher usage of the ω-k and CSA algorithms. There is also a gentle increase in packet delay with the higher production rate.

Figure 9-12 shows the reaction of the system-wide optimization strategy. Unlike the two previous experiments, the measured average packet delay follows the change in production rate fairly closely, and the execution path selections adapt to the change rapidly. This is in large part due to the fact that the sensor is always aware of its own production rate, and therefore takes this into account immediately during its calculations.

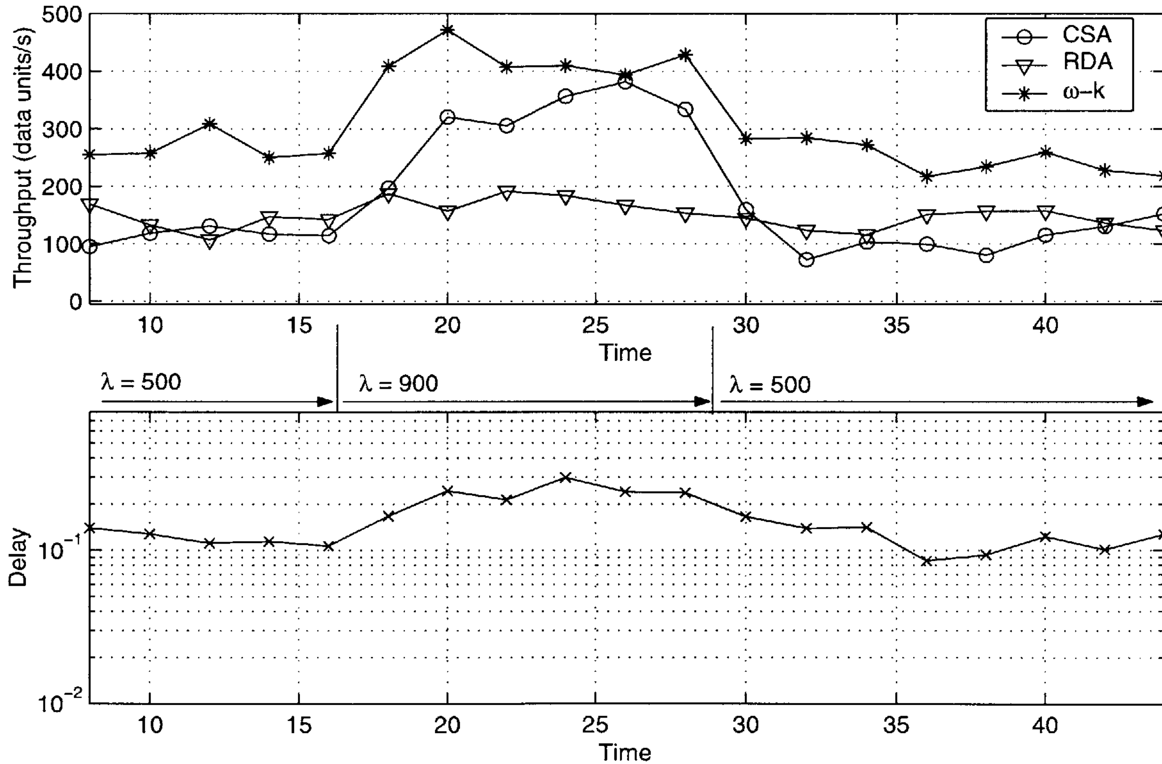


Figure 9-11: Packet-wise adaptation: Reaction to change in workload.

9.4 Summary

This chapter presents an approximate assessment of the performance of algorithm adaptation in the type of distributed processing environment under investigation in this thesis. Our simulation results

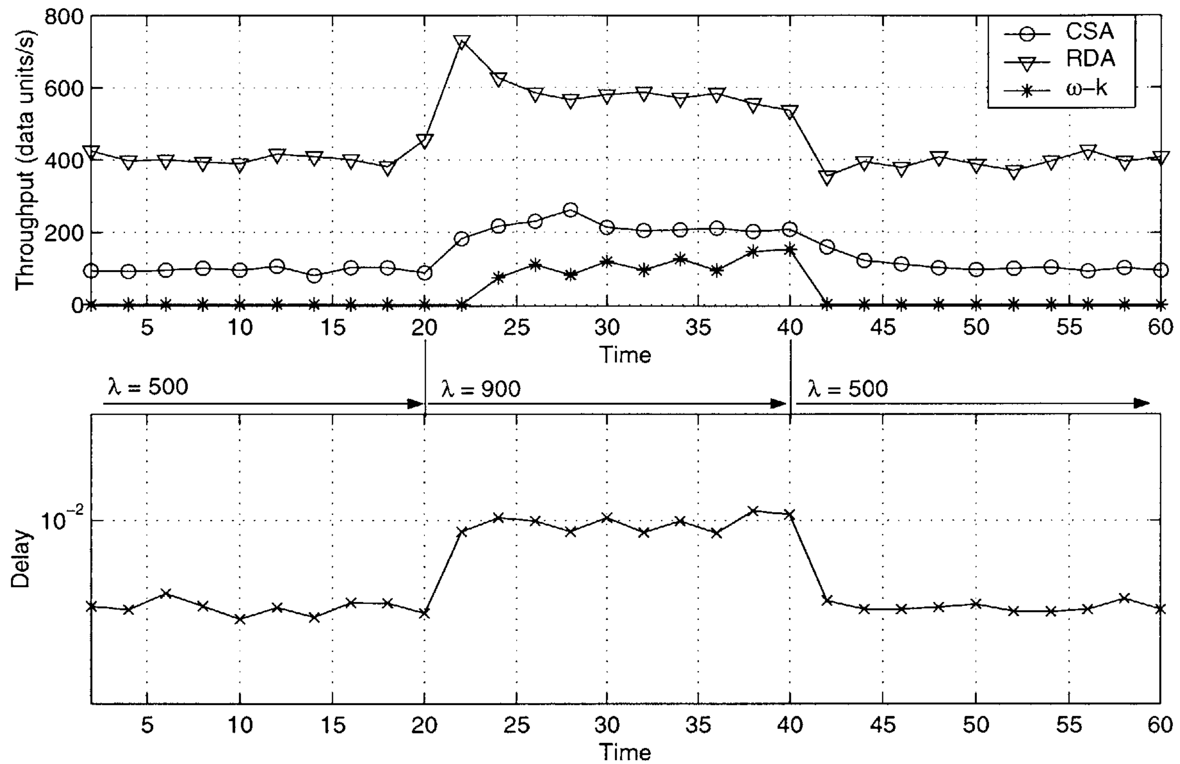


Figure 9-12: System-wide adaptation: Reaction to change in workload

demonstrate the benefits of algorithm adaptation and, in particular, system-wide adaptation. Under algorithm adaptation, the system capacity is improved, and system performance adapts gracefully to changes in the processing load and processors composition. The advantages of system-wide optimization are demonstrated by the consistently lower packet delays achieved when system-wide adaptation is in use. The promising results obtained in this chapter highly suggests that a more systematic simulation study be undertaken to firmly establish our observations.

Chapter 10

Summary and Future Directions

10.1 Conclusions

This thesis introduces a framework through which the selection among functionally equivalent signal processing algorithms can be dynamically performed by the system, rather than by the user. In dynamic, heterogeneous, distributed computing environments, this process relieves the user of the difficult problem of predicting the characteristics of the computing platform which would ultimately execute the desired program. At the same time, it gives the system an additional degree of freedom in its mechanisms of resource allocation. Our simulations show that the adaptation process enables the system to perform satisfactorily over a wide range of operating conditions.

The foundation of the framework is a model of the processing network as a collection of mutually connected specialized virtual processors. The use of this model makes this work a significant deviation from those found in the computer science literature, where it is immediately assumed that every processor is a general purpose microprocessor. Under this model, a general purpose microprocessor is simply a set of specialized virtual processors, with the elements of the set determined by the availability of software for the implementation of each functionality. At the same time, the processing limitations of highly specialized devices such as ASICs and programmable devices such as FPGAs are easily captured. The application of this model throughout the thesis demonstrates its flexibility and analytical tractability.

The overall framework is simplified considerably by the encapsulation of the data with a task description which presents the system with options on different ways to satisfy the processing requirements. Multiple equivalent implementation choices are expressed in terms of the primitive operations available in the processing network using an algorithm description language. From a practical perspective, the algebraic task descriptions permits the processing steps to be specified to the system at a level higher than machine-level code, leaving implementation details to individual hardware platforms. Moreover, the use of the data block objects clearly delineates the initiation

and termination of each processing step, allowing migration of the overall processing job from node to node in the network.

Beyond these basic advantages, the algebraic task description language is naturally represented by algorithm graphs, whose construction is noteworthy for several reasons. By allowing arcs with multiple head and tail nodes, the algorithm graphs can express concurrency and equivalence simultaneously. By pointing to an interpretation of computation as analogous to communication networks, they bring into focus many components of typical computing platforms, such as fault tolerance schemes, approximate processing strategies, and, of course, algorithm adaptation. For example, our work on the definition of system capacity with respect to given algorithm graphs is built through analogous concepts of maximum flow in standard network flow literature.

The structure of algorithm graphs suggests many possible algorithm adaptation strategies, two of which are formulated and explored in the thesis. In packet-wise cost minimization, every virtual processor is assigned a cost, and every data block is processed with the execution path which incurs the least cost. The cost associated with each virtual processor is chosen according to the current utilization level of a resource of interest, such as time or power. Like shortest path routing in communications networks, this simple strategy offers adaptive behavior but suffers from instability problems, which are avoided by using the system-wide algorithm adaptation strategy. This second strategy formulates the problem of execution path selection as a constrained optimization problem, using a cost function which relates to overall system congestion. The minimization problem is solved via a distributed version of the gradient projection algorithm.

To better quantify the effects of allowing execution paths to be dynamically chosen by the system, the thesis formulates a measure of the system capacity. For a given system of processors of known functionality and rates, the system capacity with respect to one or multiple given algorithm graphs is defined and solvable using standard linear programming codes. The capacity measure is independent of the algorithm adaptation strategy, and hence serves as an benchmark of the relative merits of different strategies. Additionally, its solution reveals the bottleneck of the system with respect to the given processing tasks. Using this definition of system capacity, we formulate the problem of functionality assignment to a set of processors in order to meet a set of processing demands. The solution to this problem is straight-forward when all processing demands are known and constant. However, when processing demands are uncertain, the solution highlights the benefits of resource sharing gained in offering the system options on the actual sequence of steps used to process data.

The construction of the simulation environment gives us a degree of practical experience in the implementation issues. We describe a set of functionally equivalent algorithms for processing data collected on stripmap synthetic aperture radars (SARs). The simulation results demonstrate some of the potential gains of the algorithm adaptation framework. Without algorithm adaptation,

the system capacity is lower, and hence the system performance is reasonable only in a narrow range of processing demands. On the other hand, both algorithm adaptation strategies result in system performance which degrades gracefully as processing load increases. Moreover, they are both able to respond to changes in the computing environment, such as processor failures or additions. Moreover, system-wide optimization results in significantly better system throughput than packet-wise optimization.

Most importantly, this thesis addresses a new problem, that of enhancing distributed computing systems to facilitate both the design and performance of signal processing algorithms executed on such systems. The framework relies on the existence of functional equivalences as well as primitive building blocks for signal processing tasks. By leveraging these two characteristics, we create a heterogeneous and dynamic distributed computing environment which operates efficiently over a broad range of operating conditions. As signal processing applications play increasingly important roles, and as distributed systems become even more widespread, it is crucial to understand the characteristics of both technologies, and how they can be mutually exploited to complement each other. It is hoped that this work represents an initial foray into such explorations.

10.2 Future Directions

Beyond the issues which are described within the thesis, our explorations in this work have revealed a myriad of other interesting research directions. In this section, we describe a subset of those which we find particularly intriguing and important to investigate.

10.2.1 System Architecture Design

The design of the architecture of a distributed signal processing system involves many complex and challenging issues. For instance, in the context of algorithm adaptation, the design and selection of the primitives of the system involve concepts from both the signal processing and the computer science perspectives. It requires familiarity with large sets of signal processing algorithms to formulate those operations whose compositions perform meaningful DSP tasks. Moreover, the design of modularized and self-contained primitive building blocks may benefit from expertise gained from object-oriented software design in the computer science area. Beyond considerations for completeness and granularity, many implementational issues, from wordlength precision to data formatting, must all be resolved.

The broader question of the type of devices which are most suited for inclusion in a distributed signal processing environment warrants further study. The tractability of the system model described in Chapter 3 suggests that it may be feasible and even preferable to build distributed signal processing systems using simple processors which are pre-programmed with common signal process-

ing functions and attached to a memory bank. Such a system architecture may be simpler to use, manage, and upgrade than more traditional networks composed of general purpose microprocessors.

Finally, systems with dynamically programmable devices such as FPGAs offer a rich area of exploration. Because these devices can be re-programmed at any time, there can be two simultaneous threads of adaptation in such systems. The first thread performs algorithm adaptation, and finds the execution paths which would minimize system cost, given the current processing demands and current functionalities of the processors. The second thread performs system adaptation, through which it reprograms the functionalities of the processors so that the system best serves the current set of processing demands. Explorations into how the functionalities of the processors can be dynamically adapted, as well as how these two optimization processes interact, would be particularly interesting and significant.

10.2.2 System Modeling

For any system architecture, an important step in the formulation and analysis of operating policies is the appropriate modeling of the system characteristics. The models used in this thesis can be enriched and made more accurate in several important directions. For example, one simple extension involves capturing the costs of communications as well as data processing. Such a model can facilitate the simultaneous optimization of both the execution path used for processing and the communication route used to send data from processor to processor.

An important improvement involves the explicit modeling of finite-sized queues at each processor, which would induce the blocking of new data processing requests and impose tighter limits on the processing capacity of the system. Existing literature in the queueing theory suggests the difficulty of an exact analysis for queueing systems of this nature. Nevertheless, advanced approximate analysis and simulation techniques may provide insights which help to quantify and predict the system behavior. Successful theoretical work in this area would significantly hasten the progress in many fields, including communications networks and manufacturing systems.

The concept of system capacity developed in Chapter 6 provides a benchmark against which the effectiveness and efficiency of a system can be measured. Realistically, the intermittent failures of processors would limit the system performance. Therefore, a statistical measure of this quantity which accounts for the effect of processor failures and recoveries would provide a tighter upper bound to the actual achievable system performance.

10.2.3 System Operation

The operational aspects of a distributed signal processing system, from the user interface to the resource allocation policy, also contain many interesting problems for further study. For example, the algorithm algebra developed in Chapter 4 is only a first step in the development of a language to

express signal processing algorithms to allow the system maximal flexibility. It should be extended to allow the expression of more complex processing tasks, by including conditional and looping constructs. Accompanying the addition of these constructs should be suitable analysis methods to be used during the optimization process of algorithm adaptation.

More generally, the design of tools to simplify the programming process is a challenging and important problem. For instance, tools to generate functional equivalences, or to identify the equivalence between different algorithms would be of great assistance to users working within the algorithm adaptation framework. Previous work in symbolic signal processing provides a sound starting point from which further studies can proceed [13].

Algorithm adaptation represents only one interesting and effective strategy for facilitating signal processing in distributed computing environments. The analogy between algorithm graphs and communications networks inspires consideration for a variety of other system level services. As an example, existing works on quality-of-service considerations in communications networks may be transferable to the computation context, allowing the system to prioritize packet processing based on its urgency, importance, or other factors. A related area is the exploration of strategies for a real-time distributed processing system. The implementation of both of these services can exploit the availability of approximate signal processing techniques, which allow computation steps to be scaled back in the event of time or power limitations [27].

Bibliography

[1] R. Bamler. "A Comparison of Range-Doppler and Wavenumber Domain SAR Focusing Algorithms." *IEEE Transactions on Geoscience and Remote Sensing.* July 1992.

[2] P.E. Beckmann and B.R. Musicus. "Fast Fault-Tolerant Digital Convolution using a Polynomial Residue Number System," *IEEE Transactions on Signal Processing.* Vol. 41, No. 7, July 1993.

[3] J.R. Bennett, I. G. Cumming, And R. A. Deane. "The Digital Processing of SEASAT Synthetic Aperture Radar Data," *IEEE International RADAR Conference,* 1980, 168–175.

[4] D.P. Bertsekas. "A Class of Optimal Routing Algorithms for Communication Networks", *Proceedings of the Fifth International Conference on Computer Communication.* Atlanta, GA, Oct. 1980, pp. 71-76.

[5] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[6] D.P. Bertsekas and R.G. Gallager. *Data Networks, 2nd Edition*, Prentice-Hall, 1992.

[7] J.R. Budenske, R.S. Ramanujan, and H.J. Siegel. "On-Line Use of Off-Line Derived Mappings for Iterative Automatic Target Recognition Tasks and a Particular Class of Hardware Platforms," in *Proceedings of the Heterogeneous Computing Workshop,* 1997.

[8] C. Cafforio, C. Prati, E. Rocca. "SAR Data Focusing Using Seismic Migration Techniques." *IEEE Trans. on Aerospace and Electronic Systems,* Vol. 27, No. 2, Mar. 1991, pp. 194–206.

[9] E. Canuto, G. Mengo, and G. Bruno. "Analysis of Flexible Manufacturing Systems," in *Efficiency of Manufacturing Systems,* editted by B. Wilson, C.C. Berg, D. French. New York, Plenum Press, 1983.

[10] W. G. Carrara, R. S. Goodman, and R. M. Majewski. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms* Norwood, MA: Artech House, 1995.

[11] T.L. Casavant and J.G. Kuhl. "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," from *Readings in Distributed Computing Systems*, IEEE Computer Society Press, 1994.

[12] M. Colajanni, P.S. Yu, and D.M. Dias. "Analysis of task assignment policies in scalable distributed web-server systems," *IEEE Transactions on Parallel & Distributed Systems.* Vol.9, No.6, June 1998, pp. 585-600

[13] M.M. Covell, C.S. Myers, and A.V. Oppenheim. "Computer-Aided Algorithm Design and Rearrangement," in *Symbolic and Knowledge-based Signal Processing*, ed. A.V. Oppenheim and S.H. Nawab. Prentice Hall, Englewood Cliffs, NJ, 1992.

[14] G.R. Dattatreya and R. Venkatesh. "Static and Decentralized-adaptive Load Balancing in a Star Cnfigured Distributed Computing System," *IEEE Tranactions on Systems, Man, & Cybernetics Part A: Systems & Humans.* Vol. 26, No. 1, Jan. 1996, pp. 91-104.

[15] A. Desrochers. *Modeling and Control of Automated Manufacturing Systems.* IEEE Computer Society Press, 1990.

[16] D.L. Eager, E.D. Lazowska, and J. Zahorjan. "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering.* Vol SE-12, No. 5, May 1986, pp. 662-675.

[17] P.H. Enslow, Jr., "What is a 'Distributed' Data Processing System?" *Computer*, Vol. 11, No. 1, Jan. 1978, pp. 13-21.

[18] M.M. Eshaghian, Y.C. Wu. "Resource Estimation for Heterogeneous Computing," *Future Generations Computer Systems.* Vol. 12, No. 6, June 1997, pp. 505-520.

[19] D. Hensgen, editor. *Sixth Heterogeneous Computing Workshop*, IEEE Computer Society Press, 1997.

[20] C. V. Jakowatz, Jr., D.E. Wahl, P. H. Eichel, D. C. Ghiglia, and P. A. Thompson. *Spotlight-Mode Synthetic Aperture Radar: A Signal Processing Approach* Boston: Kluwer Academic Publishers, 1996.

[21] M. Y. Jin and C. Wu. "A SAR Correlation Algorithm which Accommodates Large Range Migration." *IEEE Trans. on Geoscience and Remote Sensing,* Vol GE-22, No. 6, November 1984. pp. 592-597.

[22] J. Kimemia and S.B. Gershwin. "An Algorithm for the Computer Control of a Flexible Manufacturing System," *IIE Transactions* Vol. 15, No. 4, Dec. 1983, pp. 353-362.

[23] D. Klappholz and H.C. Park. "Parallelized Process Scheduling for the Support of Task Forces," *Proceedings of the International Conference on Parallel Processing.* 1984, pp. 315-321.

[24] C.H. Lee and K.G. Shin. "Optimal Task Assignmenet in Homogeneous Networks," *IEEE Transactions on Parallel & Distributed Systems.* Vol. 8, No. 2, Feb. 1997, pp. 119-129.

[25] J.-Y. Lin, L. Teng, and F. Muir. "Comparison of Different Interpolation Methods for Stolt Migration." *Stanford Exploration Project, Report 79.* November, 1997, pp. 269–276.

[26] V.M. Lo. "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transactions on Computers.* Vol. 37, No. 11, Nov. 1988, pp. 1384-1397.

[27] J.T. Ludwig, S.H. Nawab and A. Chandrakasan. "Low-Power Digital Filtering using Approximation Processing," *IEEE Journal on Solid State Circuits.* Vol. 31, No. 3, March 1996.

[28] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Francisco, 1996.

[29] P.Y.R. Ma, E. Y. S. Lee, and J. Tsuchiya. "A Task Allocation Model for Distributed Computing Systems." *IEEE Transactions on Computers,* Vol. C-31, No. 1, Jan. 1982, pp. 41-47.

[30] C. McMillan, Jr. *Mathematical Programming: An Introduction to the Design and Application of Optimal Decision Machines.* John Wiley, New York, 1970.

[31] D.C. Munson, Jr., and R.L. Visentin. "A Signal Processing View of Strip Mapping Synthetic Aperture Radar." *IEEE Transactions on Acoustics Speech and Signal Processing,* December, 1989.

[32] P.J. O'Grady and U. Menon. "A Concise Review of Flexible Manufacturing Systems and FMS Literature," *Computers in Industry.* Vol. 7, 1986, pp. 155-167.

[33] J. Orlin, R. Ahuja, and T.L. Magnanti. *Network Flows.*

[34] J. Ousterhout, D. Scelza, and P. Sindhu. "Medusa: An Experiment in Distrbuted Operating System Structure." *Communications of the ACM,* Vol. 23, No. 2, Feb. 1980, pp. 92-105.

[35] R. K. Raney, H. Runge, R. Bamler, I. G. Cumming, and F. H. Wong. "Precision SAR Processing Using Chirp Scaling." *IEEE Transactions on Geoscience and Remote Sensing.* July, 1994.

[36] A. Reinefeld, et.al. "The MOL Project: An Open, Extensible Metacomputer," in *Proceedings of the Heterogeneous Computing Workshop,* 1997.

[37] D. Ridge, D. Becker, et. al. "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs," *Proceedings, IEEE Aerospace.* 1997.

[38] P.R. Romig III and A. Samal. "DeViouS: a Distributed Environment for Computing Vision," *Software-Practice & Experience.* Vol. 25, No. 1, Jan. 1995, pp. 23-45.

[39] T. E. Scheuer and F. H. Wong. "Comparison of SAR Processors Based on a Wave Equation Formulation." *IGARSS-91* pp. 635-639.

[40] C. Shen and W. Tsai. "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Transactions on Computers.* Vol. C-34, No. 3, Marc. 1985, pp. 197-203.

[41] P.A.T. Sivarammurthy. "Optimal Task Allocation in Distributed Systems by Graph Matching and State Space Searching," *Journal of Systems & Software.* Vol. 46, No. 1, Apr. 1999, pp. 59-75.

[42] J.J. Solberg. "A Mathematical Model of Computerized Manufacturing Systems," *Proceedings of the 4th International Conference on Production Research.* Tokyo, Japan, 1977.

[43] A. Smailagic, D. Siewiorek, et. al. "MoCCA: a Mobile Communication and Computing Architecture," *Mobile Computing & Communications Review.* Vol. 3, No. 4, Oct. 1999, pp. 39-45.

[44] J.A. Stankovic. "The Analysis of a Decentralized Control Algorithm for Job Scheduling Utilizing Bayesian Decision Theory." *Proceedings of the International Conference on Parallel Processing.* 1981, pp. 333-337.

[45] J.A. Stankovic. "Distributed Computing" from *Readings in Distributed Computing Systems*, IEEE Computer Society Press, 1994.

[46] H.S. Stone. "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering.* Vol. SE3, No. 1, Jan. 1997, pp.85-93.

[47] V.S. Sunderam and G.A. Geist. "Heterogeneous Parallel and Distributed Computing," *Parallel Computing.* Vol. 25, No. 13-14, Dec. 1999, pp. 1699-1721.

[48] P.P. Vaidyanathan. *Multirate Systems and Filter Banks* Prentice Hall, 1993.

[49] Xilinx Corporation. "Xilinx DSP: High Performance Signal Processing." January, 1998.

[50] W.-L. Yang. "A Distributed Processing Architecture for a Remote Simulation System in a Multi-user Environment," *Computers in Industry.* Vol. 40, No. 1, Sept. 1999, pp. 15-22.