Modeling and Optimizing Quality for Networks of Approximate Processors

by

Matthew J. Secor

B.S., Electrical Engineering and Computer Science,
University of California, Berkeley (1994)

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1996

© 1996 Massachusetts Institute of Technology. All rights reserved.

Author: ________________________________
Department of Electrical Engineering
August 30, 1996

Certified by: ____________________________

George C. Verghese
Professor of Electrical Engineering
Thesis Supervisor

Accepted by: ____________________________
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

OCT 15 1996

ARCHIVES

LIBRARIES
Modeling and Optimizing Quality for Networks of Approximate Processors

by

Matthew J. Secor

Submitted to the Department of Electrical Engineering on August 30, 1996, in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering

Abstract

In this thesis, we investigate optimal resource allocation in certain classes of networks of interconnected approximate processors. An approximate processor is one that allows tradeoffs between the quality of the results and the resource usage of the processor. Previous work with anytime algorithms and incremental refinement structures is used to motivate and develop a method of modeling the quality tradeoffs for approximate processors and networks, extending earlier models. The thesis reviews methods developed in [4] for compiling anytime algorithms to dynamic programming algorithms in which the quality of the output of the network is optimized, subject to some set of resource constraints. These algorithms are then extended — for the case of pipeline and forward tree networks — to a situation in which the structure of the network and of its component processors may not be static. The final algorithms allow the distributed calculation of the optimal output quality and control policy, with each processor in the network only requiring computations based on its own model and information received from preceding and succeeding processors.

Thesis Supervisor: George C. Verghese

Title: Professor of Electrical Engineering
Acknowledgments

I would like to first thank my advisor, George Verghese, for remaining patiently optimistic and available to talk at all hours. He has helped me to clarify and direct my efforts as I undertook this work; if you happen to find an intelligible sentence or clearly explained idea in this thesis, he is probably responsible to some degree. I look forward to many more years of collaboration with him. Also I would like to thank Al Oppenheim for the financial support and advice he has given me over the past year and a half.

There are a number of other people with whom I have had helpful discussions. I thank them all, as every one has helped in some way or another.
## Contents

1 Introduction ........................................ 11
   1.1 Motivation ..................................... 11
   1.2 Approximate Processing ........................ 14
       1.2.1 Incremental Refinement (IR) ............... 15
       1.2.2 Compilation of Anytime Algorithms .......... 17
   1.3 Approximate Processing Applications ............ 18
       1.3.1 Resource Constrained Systems ............... 18
       1.3.2 Dynamic Environments .................... 19
       1.3.3 Dynamic Systems .......................... 19
       1.3.4 Design Applications ....................... 20
   1.4 Contributions .................................. 21

2 Quality Modeling of an Approximate Processor ..... 23
   2.1 Interruptible vs. Contract Processing ............ 26
   2.2 Quality Model .................................. 27
   2.3 Approximate Processor Examples ................. 29

3 Optimization of Quality in Simple Networks ....... 37
   3.1 Network Modeling ................................ 37
   3.2 Quality Optimization ............................ 39
   3.3 Serial Connection of Two Processors .............. 40
   3.4 Networks ....................................... 45
3.4.1 Pipeline ............................................. 46
3.4.2 Forward Tree Networks ............................ 49

4 More Complex Networks ............................... 53
  4.1 Multidimensional Networks ....................... 54
    4.1.1 Multi-criteria Dynamic Programming (MDP) .......... 56
  4.2 Distributed Local Compilation .................. 60
    4.2.1 Single Dimensional Pipeline .................... 61
    4.2.2 Single Dimensional Trees ....................... 66
    4.2.3 Multidimensional Pipelines ..................... 71
    4.2.4 Multidimensional Trees ......................... 73

5 Conclusion ........................................... 75
List of Figures

2-1 Effect of low-pass filtering on the output of an IR DFT algorithm . . . 25
2-2 Quality model $a$, with quality input $q_a$, control input $u_a$, quality output $y_a$, and performance profile $Q_a$. ........................................ 28
2-3 Approximate processor to compute FFT. ................................ 30
2-4 Performance profile of a 256-point FFT chip. ............................. 31
2-5 Performance profile of a variable length FFT processor. ............... 31
2-6 Quality model for IR DFT processor. ...................................... 33
2-7 Performance profile of an IR DFT processor. ................................ 34
2-8 Example outputs from an IR DFT processor. ............................... 35

3-1 Series connection of two approximate processors. .......................... 41
3-2 Performance profiles of the two processors in our example. ............. 42
3-3 Output quality for our serial connection of two processors. ............ 43
3-4 Output quality for our serial connection of two processors, when $u_1 + u_2 = 50$. ............................................................ 43
3-5 Optimal allocations $u_1^*$ and $u_2^*$ as $U$ varies for our serial connection of two processors. ....................................................... 44
3-6 Optimal quality output $y_2^*$ as $U$ varies for our serial connection of two processors. ......................................................... 45
3-7 Sample forward tree network. .................................................. 46
3-8 Sample pipeline network. ........................................................ 47
4-1 Resolution vs. bit depth tradeoff for a gray-scale image. ............... 55
4-2 An example of a forward tree network. ........................................ 69
4-3 Performance profiles $Q_1(\cdot)$ and $Q_2(\cdot)$. Values in the tables denote value of the quality output. ........................................ 69
Chapter 1

Introduction

1.1 Motivation

There has been increasing interest in dynamically changing, physically distributed systems in recent years, as these types of systems become more common in the real world. For example, we have seen the migration from single computers to loosely connected, well defined, local-area networks (LANs), and on to large scale, heterogeneous, wide-area, multiuser, rapidly changing networks (the Internet explosion). This expansion has been possible in part because of the development of distributed routing algorithms that are able to operate with large-scale heterogeneous networks. Also, the existence of ubiquitous communication standards and protocols has allowed the distribution and sharing of information at a scale not envisioned a few years ago. We have also seen the use of embedded digital electronic systems and digital signal processors in everything from everyday appliances such as rice cookers and microwaves to televisions and military aircraft. This penetration of digital technology into all facets of society, and the continuing trend towards large, complex systems suggests possible areas of research into the management of such systems.

A common system of this type is a network of processors, in which each processor has some particular task it must perform on its input data to produce the appropriate output data, and the overall network, as a particular interconnection of the component
processors, has another, more complex task that must be performed. In the following chapters, we will consider systems of this type, in a setting in which the structure of the network and of the processors is not necessarily static, and where the processors can only share information with other processors which they are directly connected to.

Some of the main problems apparent with these dynamic distributed systems are the lack of knowledge about the environment in which they are operating, and the dynamic nature of the structure of the system itself. This thesis concentrates on the characterization and control of a class of processors that can be designed to allow them to deal with the uncertainty in their surroundings, alone or as part of a larger interconnected network. Many of the examples and applications deal with concepts from the discipline of digital signal processing, but the basic ideas are more generally applicable. This chapter first describes a number of attributes of the systems we are interested in, and presents motivation for various aspects of the model that we eventually choose to characterize these systems. Then we give background information on work related to the class of processors we are interested in, and describe a number of possible applications for these processors. Finally, an outline of the contributions of this thesis is given to illustrate the relationship between this and the following chapters.

A number of attributes of large-scale dynamic distributed systems and their environments serve to make the application of present control and design techniques more difficult. In order to develop useful control and design methodologies for these systems, we must take into account the following attributes of the system: distributedness, a dynamic environment, and heterogeneity. Also of importance are the following attributes of a system's environment: competing objectives, and constrained resources.

**Constrained Resources** The allocation of scarce resources such as time, power, or cost is very important when trying to maximize the performance of a dynamically
varying system. This fact implies the need for sophisticated control algorithms for networks of processors, to replace algorithms that currently rely on the static functionality of the subsystems. Different applications will vary in their needs and critical resources, and the control algorithms must take this into account, by providing runtime, design-time and off-line control, centralized and distributed control, as well as heuristic algorithms for cases in which the networks become too complex. Also, the processors themselves must be designed in order to allow their functionality to change depending on the available resources. The process of redesigning algorithms has begun, but much more work needs to be done in order to make these techniques available in a wider range of areas.

**Distributedness**  Because of the physically distributed nature of these new systems, efficient communication between components becomes an issue. Latency, bandwidth, synchronization, and unreliability of the communication links all become important, and the real system is now fundamentally different from the controlled laboratory setups in which many systems are designed. Control algorithms must take these issues into account in order to ensure that the system achieves a high performance level.

**Dynamic Environment**  One of the most important aspects of a dynamic distributed system is the ability to adapt to changes in its environment. As the system evolves and changes, the component processors must be able to update their behavior accordingly to maintain the performance of the overall system. The distributed algorithms mentioned in the previous paragraph must allow for a large scale adaptivity. As systems change structurally or parametrically, the algorithms must be able to maintain a high level of performance by reconfiguring the system appropriately.

**Heterogeneity**  Another interesting feature of the large distributed systems that are developing is the heterogeneity of the components of the systems. Any scheduling or control algorithm that attempts to deal with these systems cannot make the as-
umption that all of the components are similar in construction or behavior. A useful model of these components that could be generally applicable might be an abstract model that does not rely on the specific processing done by the component but only on the attributes of the processing that affect the performance of the other components. This would allow many physically and functionally different components to be represented using the same class of models, while reducing the complexity of the model class to the point that it is a useful tool for dealing with interconnections of many components.

Competing Objectives  With a changing environment come changing objectives; if the system is constantly changing or a new user takes control of the system or of components of the system, the performance requirements may change as well. Also, with many degrees of freedom the system must be able to choose between competing alternatives. This last aspect is something that has not been investigated very deeply in the literature in this particular context. Increasing complexity and reuse of individual components of networks of processors will dictate the need for these networks to accommodate many different important (and possibly competing) aspects of the data they process.

1.2 Approximate Processing

Currently, most conceptions of information and data processing view the integrity of the information passing through the network as inviolable; all processing done by a processor must be lossless, else the entire data set will be ruined. We call these processors lossless processors and the networks that contain only such processors lossless networks. However, in some contexts, such as signal processing and transmission of video and audio, the data that is sent or processed can or should be able to withstand some deterioration along the way without catastrophically destroying the quality of the final output data. An approximate processor, which allows tradeoffs between the quality of the results and the resource usage of the processor, is what is required in
this context. Such tradeoffs are not possible with lossless processors. An approximate processing system (we will concentrate on approximate processor networks (APNs) with specific structures) is a system whose components are not all lossless processors. It is also important to realize that the notion of a lossless processor is simply a special case of the more general notion of an approximate processor.

In the next two sections, we describe current and past work that has begun to deal with both the issues involved in designing specific approximate processors (incremental refinement), and the issues involved with assembling and controlling networks of approximate processors (compilation of anytime algorithms).

### 1.2.1 Incremental Refinement (IR)

Papers that have been published in the DSP field dealing with approximate processing, [1], [2] and [3], focus on practical applications/examples and fabrication related issues. They introduce the term incremental refinement (IR) structure, which refers to a DSP algorithm for which each new allotment of running time can be used to increase the quality of the output.

Reference [1] outlines the issues of quality and cost characterization, and the idea of multiple dimensions of quality is presented. The quality of the output of the incremental short-term Fourier transform (STFT) algorithm detailed in [1] is a function of three parameters: frequency coverage, frequency resolution and the number of quantization bits. The quality function can be maximized with a particular set of constraints on the parameters and resources available.

In [2], three examples of IR structures are given, each one detailing a different direction in which to optimize resource use, namely power, time, and image resolution. The first structure is a tapped-delay-line implementation of an FIR filter in which the number of taps used to implement the impulse response can vary. This allows the filter to conserve power when a large number of taps is unneeded. It also allows the processor to work within power constraints, while still producing a usable approximation.
The second application in [2] is an IR implementation of the discrete Fourier transform (DFT) in a similar fashion to the IR STFT algorithm described above. This implementation and IR DFT algorithms that can be generated from this implementation, given constraints on costs and quality, are described in further detail in [3]. There is ongoing work within this particular area.

Finally, [2] gives an image processing example, in which the computation of a two-dimensional inverse discrete cosine transform (DCT) is accomplished through the use of an incremental refinement structure. At each stage another bit plane is added to the output image, thereby increasing the quality of the image. This would, for example, allow a decoder at the end of a video link to reduce the quantization levels at each pixel in the displayed images in order to bring computational costs (or equivalently, time) down to compensate for an increased data rate (equivalently, higher resolution).

Many areas of DSP involve the use of algorithms that may be adapted to use approximate processing techniques. Areas such as filter design, most kinds of video/audio processing and communications and estimation all have an element of approximate processing in them, even though it may not be explicitly stated. They accomplish approximate processing using many different methods, such as varying the number of coefficients in an FIR filter, altering the frame rate in video transmission, varying the number of DCT coefficients transmitted over a video link, or estimating noise power spectra in communication signals.

The cited work on IR structures begins to explore the ways in which individual processors may be designed in order to take advantage of the approximate nature of certain algorithms and requirements, and points to the need to cope with interconnections of these processors as they become more prevalent. The work also complements the results in this thesis, as it attempts to develop processors that can be used as components in approximate processor networks, and so the algorithms presented here may be used in conjunction with a network of IR structures in order to develop a practical system that can achieve some level of desirable performance.
1.2.2 Compilation of Anytime Algorithms

Approximate processing has been a topic of research for at least a decade within the artificial intelligence (AI) community. In contrast with the IR structures described above, the AI applications of approximate processing have concentrated on higher level issues such as processor interconnection [4]. The AI community has tended to think about how abstract processes such as reasoning and decision making can be formulated using approximate processing ideas.

Papers that give a good idea of the application of approximate processing concepts to AI are [5] and [6]. Of particular interest is the discussion of anytime algorithms. The evolution of anytime algorithms and approximate processing is described in the thesis of Zilberstein, [4], which also presents a compilation method for integrating networks of interconnected anytime algorithms into a single larger algorithm.

An anytime algorithm is one that will produce outputs of nondecreasing quality as the amount of time allotted to the algorithm is increased, similar to the functionality of an approximate processor. A distinction is generally made between two types of anytime algorithms. First, there are contract anytime algorithms, which come with a specification how much time they require to produce an answer; no answer is guaranteed if the algorithm is stopped before the specified time. There are also interruptible anytime algorithms, which can be stopped at any time and will produce a usable answer.

The performance of an anytime algorithm is usually given by means of a performance profile, which plots the output quality as a function of processing time. To analyze an anytime algorithm or a composition of such algorithms, only the performance profiles of the individual pieces need be known. The profiles suffice to compute an allocation of time to each of the pieces such that the output quality is maximized. The thesis [4] describes an approach for maximizing the output quality in this way for various types of interconnections of anytime algorithms; the individual algorithms are compiled together into a single new higher level contract anytime algorithm.

The result in [4] that we will be most interested in is the local compilation algo-
algorithm. We will explicitly show that Zilberstein's local compilation algorithm is simply a dynamic programming algorithm. Zilberstein does mention the fact that his algorithm is related to dynamic programming (DP), but by filling in the details we allow his results to be used in other situations where DP has been successfully applied, and enable the considerable number of DP tools that have been developed over the last four decades to be used to solve the problem at hand, namely that of optimizing the performance of networks of approximate processors. We also begin to show how his results may be extended to be usable for more complex problems, with more complex algorithms that do not fit within Zilberstein's original formulation.

1.3 Approximate Processing Applications

Approximate processing is applicable to various types of design and scheduling problems in the context of digital signal processing (DSP). The following examples are meant to provide a sample of the range of tasks for which the application of approximate processing techniques could provide a useful solution. This thesis focuses mainly on modeling, and on developing methods of resource allocation and control for resource constrained systems, and so these examples are not expanded upon, but simply used as a motivation for further investigation.

1.3.1 Resource Constrained Systems

Perhaps the most straightforward application of approximate processing to the field of DSP system design is in the area of resource constrained systems (e.g., a real-time controller that has a limited amount of time allocated to making a control decision). The development of a general framework in which to design, interconnect and control a network of approximate processors could provide for much more flexible and efficient methods of constructing and allocating resources to the various components of a system. Currently, some DSP systems are designed to meet these types of constraints for specific applications, but the resulting systems typically have the constraints fixed
at design-time, and the particular methods of dealing with the constraints are not always generalizable to other related systems. Another concern is that of allocating resources between systems which were not initially designed to inter-operate.

1.3.2 Dynamic Environments

In order to deal effectively with a dynamically changing environment, a DSP system must be able to modify its behavior to account for any changes. The use of approximate processors would allow systems to effectively respond to changes in the system's environment (e.g., fluctuating data rates, operating temperature, power supply), and make behavioral changes (e.g., lower-resolution output to speed up data throughput, slower computations to ensure accuracy, less accurate computations to conserve power) to accommodate the environmental changes. This necessitates the development of fast, run-time control algorithms that can effectively allocate the system resources in order to optimize the performance of the system. A simple example of the type of application in which this would be useful can be seen in the low-power approximate filtering application described in [2], where the stopband attenuation of a FIR filter is modified in real-time according to the current stopband energy of the input signal.

1.3.3 Dynamic Systems

Similar to the idea of a dynamic environment is that of a dynamic system, in which the actual pieces of the DSP system change over time. For example, if a system is built using a processor running at a particular clock speed and at a later time is replaced with a faster processor, the system should be able to (within reason) reallocate its resources to take advantage of the increased speed. Issues like this are also important when dealing with systems in which portions may break down and cannot be replaced immediately. In these cases, traditional DSP systems are not very useful because the architecture of the system is static, and any change will generally cause a complete breakdown of the system performance. One task for our
research is thus the development of DSP architectures that allow missing or changed functionality to be accommodated by the system. This research includes development of on-line distributed control algorithms that are able to keep the system running at a high performance level even in the presence of local breakdowns during run-time. Approximate processors lend themselves to this sort of scenario well, as they are designed with the ability to deal with changing environment and requirements.

1.3.4 Design Applications

Approximate processors can be used in the design, prototyping and operation phases of a system's development. Described below are a few applications in the areas of system design and prototyping. This thesis does not explore these applications directly, but some of the ideas developed in later chapters could be applied to some of the design problems.

Design Reuse

Approximate processors can be seen as flexible, multi-function sections of a larger design that can be used to represent different functions in a prototyping situation. This reduces the number of smaller designs needed to prototype a system, while allowing the same amount of functionality. An approximate processor can be used to simulate a low speed, high precision piece of a design, or — by simply modifying the parameters — a high speed, low quality piece that performs a similar numerical function, but for a different purpose. As described earlier, this reduces the complexity of the design process and allows the design process to go through prototyping iterations more quickly.

New Design Choices

New design possibilities are also created by the use of approximate processors; the flexibility of the processors lets them be dynamically allocated to different phases of a task. A single processor can be used to process the input data immediately after
it comes in; then, after some intermediate processing, it can reconfigure and perform different computations with the previously processed data.

**Design Tweaking**

Prototypes are often not perfect, or are used as testbeds for different designs. The use of approximate processors in the prototyping process would allow the prototype to test many different variations on the same basic design. This would also provide the opportunity to correct mistakes without rebuilding the prototype, or the opportunity to simply test out new ideas in order to get a design to work.

### 1.4 Contributions

The structure of the thesis is as follows.

This chapter, Chapter 1, provides motivation and background for the rest of the chapters.

Chapter 2 discusses the requirements of a general quality model for an approximate processor which will be able to represent a large number of interesting processors. The model that is developed is an extension of that used by Zilberstein, but includes a more general notion of quality as a multidimensional attribute of a set of data. At the end of the chapter, three examples of different types of approximate processors are given, and used to illustrate issues involved in the characterization of models for particular processors.

This leads to the consideration of networks of approximate processors beginning in Chapter 3. Although much of the chapter involves a restatement of Zilberstein’s basic results, the main contribution of the chapter is to illustrate the connection between Zilberstein’s compilation methods derived in [4] and DP algorithms. As a result of embedding the problem of output quality optimization of networks in a DP framework, we allow future work in the area to apply existing results that deal with DP problems. The end of the chapter also discusses briefly one of the assumptions
made by Zilberstein, namely that for any network of approximate processors, the total resource usage of the network is the sum of the resource requirements of the individual processors. This restriction is not necessary for either the algorithms here, or Zilberstein’s compilation algorithms, and an alternative restriction that is likely to surface when working with physically distributed networks is given. In this way, both Zilberstein’s and our models and algorithms are shown to be more general than previously depicted.

In Chapter 4, we begin to study networks that were not possible with Zilberstein’s original formulation. We investigate networks with multiple associated quality attributes that must all be optimized with respect to some set of criteria. A statement of an established multidimensional DP algorithm is given, followed ultimately by the development of a new multidimensional distributed algorithm that can be used to optimize the output quality of all of the networks discussed up to that point. This last algorithm includes as special cases all of the algorithms presented beforehand as well, and provides a much more general and more widely applicable method of quality optimization than was previously available.

The concluding remarks in Chapter 5 describe a few more concepts that are worth considering as further work in this area.
Chapter 2

Quality Modeling of an Approximate Processor

In this chapter, we develop a mathematical model for an individual approximate processor; the following chapters will concentrate on particular methods for controlling the performance of specific types of approximate processor networks using the model presented here. For the processor model to be useful in the context of networks of approximate processors, it must be rich enough to describe accurately many specific approximate processors and networks of interest, and be described simply enough for tractable design of the processors and of networks of these processors. The algorithms in Chapters 3 and 4 provide evidence that our model satisfies this last criterion, and this chapter will establish the richness of our model.

In order to allow our model to be applied to a large class of different types of processors, we will concentrate on the characterization of the approximate nature of a processor, rather than on the input-output relationship of the processor. By this we mean that the specific nature of the processing being done (e.g., FFT computation, ARMA filtering, edge detection, etc.) is not as important to the construction of a model as the processor's ability to generate outputs of varying quality from inputs of varying quality. We would like to capture the quality vs. performance tradeoffs that are possible with a particular approximate processor by encapsulating them in
a simple description called a \textit{performance profile}.

To develop a performance profile for a processor, we need to first determine which aspects of the input and output data contribute to the quality of the data. These aspects are referred to as \textit{quality measures}, and a useful choice of quality measures is quite dependent on the actual processing that is intended; there is no simple method that can be used in making the choice. For any particular application, the choice is left to the designer of the processor, and is not discussed in this thesis. It is assumed that the quality measures used in a model adequately describe the important approximate aspects of the processor, and so knowledge of the actual processing being performed is not needed when optimizing performance. This constraint points out the importance of the choice of quality measures when creating a model for an approximate processor; the wrong choice of quality measures will produce a model for the processor that does not have the ability to capture all of the relevant information.

Another problem that may surface due to a bad choice of quality measures is that the relationship between the quality of the input and the quality of the output may not be \textit{well-defined}; i.e., two different input data sets of the same quality could give rise to two output data sets of differing quality. This last point is crucial, as it is key to allowing the relationship between the input and output quality to be encapsulated in a single function, depending only on the current state of the processor and the quality of the input data in order to determine the quality of the output data.

An example of a possible problem that might arise when choosing the quality measures for a particular processor can be seen in the IR DFT implementation given in [3]. In this case, three quantities were chosen as relevant measures of quality of the output of the DFT: frequency coverage, frequency resolution, and the number of quantization bits. Thus, one might choose to generate samples of the DTFT of a discrete time sequence with a particular spacing (frequency resolution), with a certain number of bits (number of quantization bits), and within a certain bandwidth (frequency coverage). However, this particular formulation is indifferent to the location of the band; e.g., it does not distinguish between five 8-bit samples taken between
Figure 2-1: Effect of low-pass filtering on the output of an IR DFT algorithm.

0 and $\pi/2$, and five 8-bit samples taken between $\pi/2$ and $\pi$. Characterization via these quality measures could prove to be misleading if the output of the DFT is to be selectively filtered in later processing. If the output is to be low-pass filtered, leaving only the samples between 0 and $\pi/4$, then the second set of samples given above would be useless, as they would all be filtered out. These two cases are illustrated in Figure 2-1. A possible fix for this problem would be to consider four quality measures of the output of the DFT: frequency resolution, quantization bits, highest frequency, and lowest frequency.

Another possible measure of quality could be the signal-to-noise ratio of the data at the input and output of an iterative noise-reduction processor (i.e., each round of processing done by the processor increases the signal-to-noise ratio of the output data). Or the order of a filter designed using the Parks-McClellan algorithm may be used as a measure of quality; as the order increases, the filter is able to meet tighter specifications on the passband, transition band, and stopband characteristics. Quality measures may also be more detailed than what we have described. While doing processing of an image, in addition to the total number of bits in each pixel of the output image, for example, the number of bits allocated to each of the red, green, and blue components of a pixel may be needed in a context in which various observers of the image may be more sensitive to particular colors.
In addition to the restriction that the quality measures of the input and output of a processor be well-defined with respect to the processor, we assume for simplicity that the input and output data streams of a processor can be divided up into non-overlapping sections such that each section of the output data stream is a function of exactly one section of the input data stream. If this is the case, we may say that the quality of one of these sections of the output data stream is a function only of the quality of the corresponding section of the input data stream and the state of the processor itself. Thus, we can determine the measure of the quality of each of these sections of the input data stream (or output data stream) independently.

An example of a system that has this property would be an image transmission system that takes as input an entire image and codes the image independently of any previous or future images, then transmits and decodes the image at another location. Quality measures in this situation would describe the quality of a single image and would normally vary from image to image. If we consider a video transmission system that also employs predictive coding, using a number of previous images to predict the next image, and if we still use quality measures on each individual image, then the system would have memory and thus the quality measure for one image would depend on the quality of other images, and we would not be able to model the system accurately using the model developed below.

2.1 Interruptible vs. Contract Processing

As already noted (in Section 1.2.2), there are two useful types of approximate processors, namely interruptible and contract processors. An interruptible approximate processor can be stopped at any point in its processing and will return a usable output. It does not need to be given a resource allocation to begin processing, but will incrementally increase the quality of its outputs as it consumes more resources. On the other hand, contract approximate processors demand a specific resource allocation before they begin processing. If they are stopped before the entire contracted
allocation has been used, their outputs will have no guarantee of any level of quality. However, after the allocation has been used, their outputs will achieve at least the level of quality given by the performance profile. In this thesis, we will deal mainly with contract approximate processors, as they allow particular techniques to be used in order to optimize the output quality of networks. As a result, many of the techniques below cannot make use of the additional flexibility given by interruptible processors. Contract approximate processors have the nice feature that they may be emulated using a processor that implements a class of non-approximate algorithms. For example, a processor that can generate the $n$-point FFT of an $N$-point input sequence (with $n$ between 1 and some large number $N$) is effectively a contract approximate processor. For any particular time allocation, it can calculate the maximum length FFT that it can compute in that time interval, then use the corresponding FFT algorithm. Many current digital signal processors have the capability to implement many different algorithms within a particular class, and thus have the ability to be used as contract approximate processors within the framework given here.

2.2 Quality Model

Our quality model for an approximate processor consists of four parts: an input quality vector or quality input $q \in Q$, a control input or control $u \in U$, an output quality vector or quality output $y \in Y$, and a performance profile $Q$ which is a function $Q : U \times Q \mapsto Y$. The inputs and output will generally be real valued vectors or members of finite sets. A block diagram of such a model is shown in Figure 2-2.

The quality input and output can be thought of as measures of the quality of the input data set at the actual processor input and corresponding output data set at the actual processor output. Quality can be thought of in an intuitive sense, as a measure of how “good” a particular set of data is. More formally, quality can also be seen as a mapping from a particular data set into some member of the set $Q$ or the set $Y$. In general, the mapping is highly processor dependent, and defines the relationship
Figure 2-2: Quality model $a$, with quality input $q_a$, control input $u_a$, quality output $y_a$, and performance profile $Q_a$.

between the abstract model and the actual processor specifics. For example, it could be the mapping from an image to a vector containing the number of bit-planes and resolution of the image. These numbers can then be used as measures of quality of the image data without knowing anything about the actual image. The quality of the output data depends both on the quality of the input data as well as the particular control input $u$ that is applied.

The control inputs dictate the actual processing that takes place. They can represent many different things, such as variable parameters or resource allocations. Control inputs may even coincide with quality measures. For instance, one may choose to control the length of an FIR filter directly through a control input, while also using the same quantity as a measure of the quality of the output data.

A performance profile is simply the functional relationship (with the control input as a parameter) between the quality of the input data set and the quality of the output data set. This model assumes that this relationship is constant and well-defined as described earlier, i.e., two different input data sets with the same quality measure give rise to two output data sets with the same quality measure when the same control inputs are used.

If we assume (as all of the succeeding chapters will), that $Q \subseteq [0, 1]^p$, $U \subseteq \mathbb{R}_+^m$ (i.e., an $m$-vector whose elements are real numbers greater than or equal to 0), and $Y \subseteq [0, 1]^n$, then we have a somewhat more concrete picture of what the model is
describing. Each of the elements of a quality vector (either input or output) is called a \textit{quality measure}. Quality measures can vary from 0 to 1, with 0 being the lowest quality (signifying completely unusable data), and 1 being the highest quality (signifying perfectly uncorrupted data). Data is assigned a quality of 1 when the processor that generated it produced the best values possible in the absence of constraints on resources.

We call a model \textit{single dimensional} if $q, y \subseteq [0, 1]$, i.e., the quality input and output are scalars. A model is \textit{multidimensional} if either the quality input or quality output is multidimensional, i.e., $q \subseteq [0, 1]^p$ or $y \subseteq [0, 1]^p$, with $p > 1$. The performance profiles, as described so far, need not have any particular structure or properties associated with them. However, to simplify the analysis of the various problems in subsequent chapters, we assume that they are \textit{monotonic} in their quality inputs. A function $f(x)$ is said to be \textit{monotonic in $x$} if $f(x_0) \geq f(x_1)$ when $x_0 \geq x_1$. If $x_0$ and $x_1$ are vectors, $x_0 \geq x_1$ if every element of $x_0$ is greater than or equal to the corresponding element of $x_1$. A function $f(x)$ is said to be \textit{strictly monotonic in $x$} if $f(x_0) > f(x_1)$ when $x_0 > x_1$ (i.e., $x_0 \geq x_1$ and $x_0 \neq x_1$). So, for a performance profile $Q(u, q)$,

$$Q(u, q_0) \geq Q(u, q_1) \quad \text{if } q_0 \geq q_1.$$  \hfill (2.1)

This may seem to be somewhat restrictive, but in a practical setting, it makes sense to impose this, as any increase in input quality should never decrease the output quality. It is left to the designer of the model to ensure that this restriction holds.

\subsection{2.3 Approximate Processor Examples}

Now we will give a few examples of possible approximate processors. The first two examples are characterized by the model shown in Figure 2-3. All three examples attempt to compute some number of samples of the DTFT of an input sequence using either the FFT algorithm or the IR DFT algorithm.
Figure 2-3: Approximate processor to compute FFT.

**Example (Fixed Length FFT Processor)** First, consider a chip that can calculate a 256-point FFT of a 256-point input sequence. Suppose the quality measure we care about is the number of points in the output sequence, as a fraction of 256. If the time taken for a multiplication is 2.5\(\mu\)s and the time taken for an addition is .5\(\mu\)s, then the chip would require around \(6 \times 256 \log_2 256\)\(\mu\)s, or 12.29ms in order to calculate the entire 256-point FFT\(^1\). Intermediate results gained from the FFT algorithm before the entire sequence is computed would generally be of no use. This chip would be considered a non-approximate processor because it is designed to implement a single algorithm and takes a set amount of time to complete it. The corresponding performance profile is shown in Figure 2-4.

**Example (Variable Length FFT Processor)** Next consider a processor which can generate the \(n\)-point FFT of a 256-point input sequence. In this case, if the processor is told it only has 4ms to compute the output, it will compute the 83-point FFT, which can be done in 4ms\(^2\). In this way, it can return an output that is somewhat useful, even if it is not exactly the required output. The performance profile of this processor is represented in Figure 2-5. A few operating points are indicated with dotted lines in the figure, indicating possible configurations for the processor under different time constraints.

We see from the first example that quality models for nonapproximate processors can be represented as special cases of the general quality model for approximate

\(^1\)This calculation uses the approximation that an \(n\)-point radix-2 FFT requires \(\frac{3}{2} \log_2 n\) complex multiplications and \(n \log_2 n\) complex additions. In this case, the time requirement comes to \(6n \log_2 n\)\(\mu\)s.

\(^2\)The expression for the time needed for FFTs whose lengths are not powers of 2 is not exactly \(6n \log_2 n\)\(\mu\)s to compute, but this is a good enough approximation for this simple example.
Figure 2-4: Performance profile of a 256-point FFT chip.

Figure 2-5: Performance profile of a variable length FFT processor.
processors presented above. Thus, nonapproximate processors can be interconnected with approximate processors with no more difficulty than interconnecting approximate processors. Both of these examples illustrate the point that current processors that have not been specifically designed to be used in an approximate framework may be simply integrated; thus providing a level of backward-compatibility with existing older technology.

Another point that should be made with respect to the two performance profiles given above is that although the control input is a continuous input (time) in both cases, the control can also be an input that takes a number of discrete values. For instance, a processor may have a small number of operating modes which affect the performance profile, such as a video coder that can elect to use a number of different coding techniques. If the control input is a selector that chooses the operating mode, the performance profile will consist of a small number of functions, each of which relate the quality input to the quality output for a different value of the control input.

What if not only the number of samples, but also the position of these samples, is important? In this case, it might be more advantageous to use the IR DFT implementation given in [3]. The next example shows the steps that may be needed in order to take an existing algorithm (in this case one that has been designed as an approximate algorithm) and develop a useful quality model for it.

**Example (IR DFT Performance Profile)** Assume that we are interested in the samples $2\pi k/256$ for $k = 0, 1, \ldots, 255$ of the DTFT of a sequence $x[n]$, with the samples being computed in order of increasing $k$. If we use the IR DFT implementation given in [3] to compute the samples, we now have two measures of quality: the number of quantization bits $b$ (up to 32), and the number of samples computed $n$ (up to 256); the resolution quality measure is now restricted to be 1 because of the order in which we chose to compute the samples. A block diagram of the quality model is shown in Figure 2-6. Using an approximation of the expression in [3] for the number of complex additions needed to compute the DFT using the IR implementation, and using the assumption that $q_n = q_b = 1$, we have
that the total number of complex additions approximately equals \((256y_n)^2(32y_b)/2\), where \(y_n\) is the number of samples as a fraction of 256 and \(y_b\) is the number of quantization bits as a fraction of 32. Now, for any allocation of time \(u\), we have a set of possible ways we can compute the DFT, each of which satisfies the equation

\[
\tau_+ (256y_n)^2(32y_b)/2 \leq u ,
\]  
(2.2)

where \(\tau_+\) is the time needed to perform one complex addition. The problem with this is that we cannot write down a closed-form expression for the performance profile of this processor. Because a single input maps to more than one possible output, the relationship between input and output is not a function. Thus, we must refine the control that we have over the processing that is done. One possibility is to restrict the processor such that the ratio between \(y_n\) and \(y_b\) is some constant whose value is governed by a control input \(u_r\). This will allow the same set of output quality possibilities, by selecting the proper value of the ratio \(u_r\) in addition to a value for \(u\). Notice that the performance profile is not a monotonic function of \(u_r\) (as can be seen in the following equations); it need only be a monotonic function of the quality inputs to the model. Now we have

\[
\tau_+ (256u_r y_b)^2(32y_b)/2 \leq u ,
\]  
(2.3)

and thus,

\[
y_b \leq \frac{1}{256} \min \left(1, \sqrt[3]{\frac{16T}{\tau_+ u_r}} \right)\)  
(2.4)
Figure 2-7: Performance profile of an IR DFT processor.

\[ y_n = u_r y_b \leq \frac{1}{256} \min \left( 1, \sqrt{\frac{16 u_r T}{\tau_+}} \right). \]  

(2.5)

This final formulation gives us a functional form for the performance profile, which is shown in Figure 2-7. The two plots simply depict the quality outputs (on the z-axes) as they vary with different control inputs.

After a cursory inspection of the plots, we can see that there is a tradeoff between \( y_n \) and \( y_b \); for a fixed value of \( u \), as \( u_r \) increases, \( y_n \) increases and \( y_b \) decreases, and as \( u_r \) decreases, \( y_n \) decreases while \( y_b \) increases. In order to demonstrate this tradeoff, Figure 2-8 shows three possible outputs of the processor, with a different ratio \( u_r \) for each case. The dotted line on each plot indicates the exact DTFT between 0 and \( \pi/4 \) of some input sequence \( x[n] \). The other samples on the plots show the values output from the IR DFT processor when the sequence \( x[n] \) is input to the processor. In each case, the processor is allowed 400ms to calculate its outputs (i.e., control input \( u \) is set to 400ms) and \( \tau_+ = 1 \)ms. In the left plot, \( u_r \) is set to 3.5; the resulting output quality values given by Equations 2.4 and 2.5 are \( y_b = 1/32 \) and \( y_n = 28/256 \), i.e., the processor will compute 28 1-bit output samples, which are shown overlaid on the plot. Each sample value may be either 0 or 1. In the center plot, \( u_r \) is reduced to 0.125, so the number of bits will equal the number of
output samples. Here we see that the processor computes many fewer samples (only 9), but each sample contains 9 bits, and so the sample values are much closer to the ideal sample values of the DTFT that they correspond to. In the right plot, $u_r$ is equal to 0.65, and each of the 16 samples computed by the processor in this case has 3 bits. The sample values are much better than in the left plot, but worse than the values in the center plot. Also, there are fewer samples than in the left plot, but more than in the center plot. These three cases begin to illustrate the tradeoffs that are possible when a quality model has multiple quality and control inputs and/or multiple quality outputs. Issues related to quality optimization of these multidimensional quality models when the processors are connected into networks are investigated in Chapter 4.

These examples give a feel for the types and behavior of individual approximate processors, but a model of an approximate processor is not very useful by itself. Chapter 3 begins to investigate methods of optimizing the output quality of the model by proper selection of the control inputs. The components of the model developed in this chapter will be used in optimization techniques in the next two chapters; Chapter 3 considers interconnections of simpler quality models like the FFT examples,
and Chapter 4 extends to interconnections of more complex models similar to the IR DFT processor example.
Chapter 3

Optimization of Quality in Simple Networks

3.1 Network Modeling

In this chapter, we will study simple networks of approximate processors, and specifically how to optimize the output quality with respect to some set of criteria, using the quality model we have already developed. First, we must describe a model for an approximate processor network (APN). An APN model consists of a collection of quality models of approximate processors, connections between pairs of these models, and a set of possible quality inputs, control inputs, and quality outputs of the network. With this simple description, we see that a model for an APN is very similar to a model for an individual approximate processor. Each has quality inputs and outputs, and control inputs. In addition, the algorithms described in this chapter will allow us to establish methods of determining a performance profile for an APN from the individual performance profiles of its components. So, any of the component processors in an APN may itself be implemented via a network.

The first two parts of the above definition are exactly analogous to a typical description of a directed graph, with the set of individual quality models taking the place of the set of nodes of the graph, and the set of connections between models taking
the place of the set of arcs. Similar to our treatment of models in Chapter 2, we will refer to a network made entirely of single dimensional models as a single dimensional network, and refer to a network that contain at least one multidimensional model as a multidimensional networks. To simplify the notation, we will generally denote the individual quality models by the positive integers $1, 2, \ldots$, and the connections by pairs of the integers $1, 2, \ldots$ (e.g., a connection from the output of model $i$ to the input of model $j$ is denoted $(i, j)$). The connections show how the data flows between processors; a processor may receive input data from two different processors, in which case there would be two connections terminating at that processor’s model. Or a processor may send output data to two other processors, in which case there would be two connections starting at that processor’s model. A connection can connect all or some portion of the individual output quality measures of one model to the same number of individual input quality measures of another model. Thus, although each element of the quality output $y$ of some model may be connected to some quality input of another model, the elements may be connected to distinct models. We will always assume that any output (input) quality measures that are not connected to some input (output) are quality outputs (inputs) of the network. The control inputs to a network are simply the collection of all the individual control inputs to the individual models. The set of control inputs is constrained such that each control input is in a set $U_N \subseteq U_1 \times U_2 \times \cdots \times U_N$. This allows the network to place additional restrictions on the possible set of inputs to the network, in order to take into account global restrictions that cannot be captured at a lower level.

As an example of an APN with all of these ingredients, Figure 3-7 in this chapter shows a network consisting of five models. Here $q_1$, $q_2$, and $q_3$ are quality inputs to the network, $y_5$ is the only quality output of the network, and $u_1$ through $u_5$ are all control inputs to the network. In this case, model 3 has a two-dimensional quality input whose components are connected to the outputs of models 1 and 2.
3.2 Quality Optimization

This setup leads to a natural optimization problem: we would like to determine choices for the control inputs to the network such that we achieve some optimal, or at least desirable, values at the quality outputs, given particular values at the quality inputs. However, if we have more than one output, or multidimensional outputs, we must determine what "desirable" means. Chapter 4 discusses this in more detail. For the remainder of this chapter, we will assume that every processor in any network has a single dimensional output quality vector and a single dimensional control input. When multiple approximate processors with conforming inputs and outputs are interconnected into larger networks, the resulting system (and corresponding optimization problem) can become somewhat complicated, with multiple inputs, controls, and outputs. In some cases, such as the ones described in this chapter, however, it is rather straightforward to write down a corresponding optimization problem that allows one to maximize the output quality.

In addition to the restrictions imposed in Chapter 2 (namely, \( u_i \in \mathbb{R} \) and \( q_i, y_i \in [0, 1] \)), we will assume for simplicity and concreteness that

\[
\mathcal{U} = \{(u_1, u_2, \ldots, u_N) \mid 0 \leq \sum_{i=1}^{N} u_i \leq U, u_i \geq 0\}.
\]  

Equation (3.1) is equivalent to the situation considered in [4], where the only resource of interest is processing time. It is also applicable in other situations with resources such as cost, power, and chip area. However, it is important to realize that our framework so far does not inherently impose this restriction, and that in some situations, such as when the processing is being done in parallel by the component processors in the network, Equation (3.1) does not hold. One such case is described in more detail at the end of this chapter, and revisited in an example in the next chapter.

In the case where the approximate network consists of a single processor, the above model suggests the following optimization problem

**Problem 3.1 (Maximum Output Quality — Single Processor)** Determine \( u^* \) such
\[ y^* = Q(u^*, q) = \max_{u \in \mathcal{U}} Q(u, q) . \] (3.2)

This is simply a function maximization problem that can be accomplished by any of a multitude of established algorithms, given knowledge of the particular form of \( Q \), and \( \mathcal{U} \). For instance, if \( Q(u, q) = gau \) where \( a \) is a \( 1 \times m \) constant vector (allowing \( m \) control inputs for this example), and \( q \) is a real scalar (\( p = 1 \)), with \( \mathcal{U} = \{ u \mid Bu \geq 0 \} \) for some constant \( r \times m \) matrix \( B \), we have a linear programming problem. These relatively simple maximization problems will be present as components of the larger problems we wish to solve, but we do not focus on their solution as a large number of solution methods are readily available. We want to focus instead on optimization problems that arise when multiple processors are interconnected.

The remainder of this chapter will build up optimization methods meant to maximize the output quality of certain types of networks. The methods presented are directly based on ones presented by Zilberstein in [4], so the next sections serve mainly to place his results in a specific optimization context, and show how they correspond with ideas in dynamic programming. After this, in the next chapter we present networks with more elaborate structure and with behavior that cannot be dealt with using Zilberstein’s methods, and develop methods that are useful in dealing with such networks.

### 3.3 Serial Connection of Two Processors

Consider a pair of approximate processors connected serially. Their quality models each has a one-dimensional quality input, control input and quality output as shown in Figure 3-1. The connection is such that \( q_2 = y_1 \). Also assume that both \( u_1 \) and \( u_2 \) are real scalars such that \( 0 \leq u_1 + u_2 \leq U \), and that \( q_1 \) is some known constant in the interval \([0, 1]\). If we would like to maximize the quality of the output data from
Figure 3-1: Series connection of two approximate processors.

In this system, we must determine the optimal control inputs \((u_1^*, u_2^*)\) such that

\[
y_2^* = Q_2(u_2^*, Q_1(u_1^*, q_1)) = \max_{0 \leq u_1 + u_2 \leq U} Q_2(u_2, Q_1(u_1, q_1)).
\] (3.3)

To solve Equation (3.3) in general, we can reformulate it as follows. Define

\[
J(x) = \max_{0 \leq u \leq x} Q_2(u, Q_1(x - u, q_1)), \tag{3.4}
\]

and then simply calculate the maximum value that \(J(x)\) takes on over all \(x\) such that \(0 \leq x \leq U\). This method of solving the problem begins to point us toward the more general method of dynamic programming, introduced in the next section, as a way of dealing with the complex optimization required.

**Example (Serial Connection of Two Processors)** To help illustrate this problem, consider a serial connection of two processors as in Figure 3-1, such that the performance profiles are given by

\[
Q_1(u_1, q_1) = q_1 \left(1 - \exp\left(-\frac{u_1}{10}\right)\right) \quad u_1 \geq 0, \text{ and}
\]

\[
Q_2(u_2, q_2) = q_2 \left(\frac{u_2}{50}\right) \quad 0 \leq u_2 \leq 50,
\]

and shown in Figure 3-2. The control inputs \(u_1\) and \(u_2\) correspond to the time that is allocated for each processor to run. First, we calculate the maximum output quality and the values of the controls \(u_1\) and \(u_2\) for \(U = 50\) when the input quality \(q_1 = 1\). In this case,
Figure 3-2: Performance profiles of the two processors in our example.

we can write the output quality $y_2$ explicitly as a function of our control inputs:

$$y_2(u_2, u_1) = Q_2(u_2, Q_1(u_1, 1)) = \left(1 - \exp \left\{ -\frac{u_1}{10} \right\} \right) \left(\frac{u_2}{50} \right). \quad (3.5)$$

This is the function that we intend to maximize, and a plot is shown in Figure 3-3. Now, to find the maximum output quality and time allocations, we simply compute $y_2^*$. Because $y_2(\cdot)$ is a monotonic function each of its arguments, we would expect that the maximum would occur at $u_1^* + u_2^* = 50^1$. Thus, we can reduce the maximization of $y_2(\cdot)$ to a one dimensional problem (where $u = u_1$),

$$y_2^* = \max_{0 \leq u \leq U} \left(1 - \exp \left\{ -\frac{u}{10} \right\} \right) \left(\frac{50 - u}{50} \right). \quad (3.6)$$

A plot of this function is shown in Figure 3-4. Numerically calculating the maximum of the argument on the right side of Equation (3.6) gives the solution $y_2^* = 0.54$, $u_1^* = 15$, and $u_2^* = 35$. This solution meets with our expectations, as the second processor requires more time to reach a similar level of quality as the first processor. If we are to maximize

---

$^1$If this were not the case, we could always add something to either $u_1$ or $u_2$ in order to make the sum equal to 50, and in the process, we would increase the value of $y_2(\cdot)$. 

42
Figure 3-3: Output quality for our serial connection of two processors.

Figure 3-4: Output quality for our serial connection of two processors, when $u_1 + u_2 = 50$. 
Figure 3-5: Optimal allocations $u_1^*$ and $u_2^*$ as $U$ varies for our serial connection of two processors.

the overall quality, it seems reasonable that both processors should be operating at similar levels of quality so that neither one acts as a bottleneck. With this solution, we are able to configure the component processors of the network in order to get the best quality output possible. If one were to use the simple approach of dividing the available time equally between the two processors, the output quality $y_2$ would be equal to 0.46, about a 15% drop in quality from optimal; in many applications, the additional quality afforded by using the optimal controls might be critical for success.

What about the relationship between the optimal control inputs and the total available time $U$? Figure 3-5 shows how $u_1^*$ and $u_2^*$ vary as a function of $U$. Even for this simple example, it is difficult (perhaps impossible) to obtain closed form expressions that describe the dependence of $u_1^*$ and $u_2^*$ on $U$, and the plots in Figure 3-5 show that the dependence is intuitive at a gross level (e.g., both increase with increasing $U$), but the particular shape of the dependence is nonlinear and not immediately intuitive. It is this dependence that we would like to be able to determine, the relationship between the optimal controls and the constraints on their values, and use to optimize the quality of the outputs of APNs.

One more piece of information we would like to get from this setup is the relationship between $y_2^*$ and $U$. A plot of this can be obtained by evaluating $y_2^*$ for various values of $U$,
Figure 3-6: Optimal quality output $y_2^*$ as $U$ varies for our serial connection of two processors.

and is shown in Figure 3-6. The reason this plot is useful is that it effectively gives us the performance profile for the overall network. Just as in a performance profile for a single quality model, Figure 3-6 shows the best output quality possible given a particular control input $U$ and a given quality input. This idea will be revisited later when we deal with more complex networks.

3.4 Networks

We now look at a generalization of the two-processor network considered in the previous section. Under our constraints on the dimension of quality and control inputs and quality outputs, the graph describing the network (with processors as nodes, input-output connections as arcs) constitutes a forward tree or convergent tree, in which there is only one model with an unconnected quality output (in each connected component of the network). If this processor is taken as the root of a tree, the other processors become the internal nodes and leaves of the tree. Any inputs left unconnected are considered inputs to the overall system, and the one unconnected output
is the output of the overall system. An example of a forward tree network is shown in Figure 3-7. The reason we restrict ourselves to this particular structure is that it allows us to describe the optimal output quality of a model in terms of the quality of its inputs and its performance profile. More complex networks with diverging paths require more complex algorithms in order to optimize the output quality. However, the forward tree structure is a useful model for a large number of situations in which a single task must make use of multiple sources of input data.

In this chapter, we assume that each processor takes a single dimensional control input; in the next chapter, we give reasons why this setup is not more difficult to handle than one that allows networks with multidimensional controls for individual processors.

### 3.4.1 Pipeline

First, let us revisit the serial connection of processors from Section 3.3, i.e., each processor has a single dimensional input as well as a single dimensional output. Now, however, we will allow more than two processors to be connected in this way. We call this particular structure a *pipeline* or *pipeline network*. The structure is very simple and is shown in Figure 3-8. Now, the problem of maximizing the output quality
is easily seen to be solved by the basic dynamic programming algorithm, which is equivalent to Zilberstein's local compilation algorithm for this case.

**Problem 3.2 (Maximum Output Quality — Pipeline)** Determine the optimal policy

\[ \pi^* = (u_1^*, u_2^*, \ldots, u_N^*) \] such that

\[
y_N^*(q_1) = Q_N(u_N^*, Q_{N-1}(u_{N-1}^*, \ldots Q_1(u_1^*, q_1)))
= \max_{u \in U} Q_N(u_N, Q_{N-1}(u_{N-1}, \ldots Q_1(u_1, q_1))).
\]

If we define a state variable \( x_i = \sum_{k=1}^{i} u_k \), then the following forward DP algorithm can be used to solve Problem 3.2.

**Algorithm 3.1 (Maximum Output Quality — Pipeline (DP))** Solve the following equations

\[
J_i^*(x_i) = Q_i(u_i^*(x_i), J_{i-1}^*(x_i - u_i^*(x_i)))
= \max_{0 \leq u_i \leq x_i} Q_i(u_i, J_{i-1}^*(x_i - u_i)),
\]

starting at \( i = 1 \) and continuing through \( i = N \), subject to the conditions

\[
J_0^*(x_0) = q_1,
\]

\[
x_i \leq U,
\]
\[ x_i \geq 0, \]
\[ x_0 = 0. \]

Zilberstein's local compilation algorithm corresponds exactly with Algorithm 3.1, and he has established that it computes the optimal quality if each performance profile \( Q_i(\cdot) \) is monotonically increasing as a function of \( q_i \).

This algorithm will compute \( J_N^*(x_N) \), which describes the maximal output quality as a function of \( x_N \), the sum of all of the control inputs. The maximum value, \( y_N^* = \max_{0 \leq x_N \leq U} J_N^*(x_N) \), is the maximum output quality attainable with the given system. After \( y_N^* \) has been computed, the optimal set of controls that achieves this output quality can be computed by starting with \( x_N^* \) and \( u_N^*(x_N^*) \) which maximize \( J^*(x_N) \), and calculating \( x_{N-1}^* = x_N^* - u_N^*(x_N^*) \). This step is then repeated until all of the \( u_i^* \) have been calculated.

The basic idea behind the algorithm stated above is as follows. If we know the maximum value that the quality output of model \( i - 1 \) takes when a total \( x_{i-1} \) units of the resource have been allocated to models \( 1, \ldots, i - 1 \), then to find the maximum value of the quality output of model \( i \) for some total allocation \( x_i \), we simply need to maximize over the allocation \( u_i \) that is given to model \( i \). It is a simple but powerful algorithm that allows a complex, \( N \)-dimensional maximization to be stated in terms of \( N \) 1-dimensional maximization problems. This is the approach that we will take for all of the networks that we consider; reducing a complex optimization problem to a number of smaller, simpler optimization problems. The assumption that each performance profile \( Q_i(\cdot) \) is monotonic in its quality input argument is important for the DP approach. This is required because the control inputs that we choose must obey Bellman's principle of optimality, which in this case means that if we have an optimal set of controls for a network, \( \{u_1, \ldots, u_N\} \), that maximizes the quality output \( y_N \), the subset \( \pi_i = \{u_1, \ldots, u_i\} \) must maximize the quality output \( y_i \) for all \( i < N \). If a performance profile for model \( i \) were not monotonic in its quality input, there
would be a set of controls \( \mu_{i-1} = \{\mu_1, \ldots, \mu_{i-1}\} \) for some value of \( x_{i-1} \) that would give an output \( y_{i-1}^\mu < y_{i-1}^\sigma \) but give rise to \( y_i^\mu > y_i^\sigma \) at the output of model \( i \) for some value of \( x_i \). If all of the other performance profiles were monotonic, this would give rise to a set of optimal controls that has the suboptimal set of controls \( \mu_{i-1} \) as a subset, violating the principle of optimality. This causes problems when we attempt to compute \( J_i^*(x_i) \); because we only have available to us the maximum value of \( y_{i-1} \) for each \( x_{i-1} \) from the maximization at step \( i-1 \), we will never consider the case when \( y_{i-1} = y_{i-1}^\mu \) and thus would not find the optimal set of controls and the corresponding output quality.

The function \( J_N^*(x_N) \) that is computed using this algorithm, as in the two processor example earlier, provides us with a method of computing the performance profile of the network, i.e., we can compute the best output quality given a particular value for \( x_N \), and quality input \( q_1 \) (\( J_N^*(\cdot) \) is implicitly a function of \( q_1 \), which is used as the initial condition in the algorithm). This algorithm then can be used to reduce an APN to a single contract approximate processor which can itself be embedded in a larger APN. This is a very important observation, as it indicates that the algorithms and models presented in this thesis can be used at many different scales simultaneously without modification, from very low-level (even chip-level) interconnections to high-level systems with satellites and airplanes as components.

### 3.4.2 Forward Tree Networks

Zilberstein also extends his local compilation algorithm to apply to forward tree networks, which corresponds to relaxing our restriction that there only be a one-dimensional input to each processor in the network. We present the following algorithm as a method of calculating the maximal quality and optimal policy for this structure. Zilberstein presents a proof of the optimality of local compilation for a complete binary tree, but the results and methods he uses apply equally well to a tree with any number of branches per node. He also sets up a modified version of his local compilation algorithm that can be used to compile forward trees. A more
detailed description of that algorithm is now given.

**Algorithm 3.2 (Maximum Output Quality — Forward Tree (DP))** Given an APN

(a) Choose a processor model $g_0$ that is a component of the APN such that $q_{g_0}$ is an input to the APN.

Repeat the following steps until the root node is reached (whose output is the output of the entire network), where $u^*_g(x_{g_k})$ is the value of the control input when the output quality of model $g_k$ for a particular value of $x_{g_k}$ is maximized, and $k$ is the iteration number:

(b) Calculate

$$J^*_g(x_{g_k}) = \max_{0 \leq u_{g_k} \leq x_{g_k}} Q_{g_k}(u_{g_k}, J^{g_k}) ,$$

where $J^{g_k}$ is the set of $J^*_j(x_j)$ such that model $j$ is connected to the input of model $g_k$. This maximization must be done over all of the $x_j$ such that $u_{g_k} + \sum_j x_j = x_{g_k}$. The values of $x_j$ that attain the maximum quality for a particular value of $x_{g_k}$ are denoted by $x^*_j(x_{g_k})$.

(c) Choose $g_{k+1}$ such that either:

- For each connection $(j, g_{k+1})$, $j = g_\ell$ for some $\ell \leq k$, or
- $q_{g_{k+1}}$ is an input to the APN.

with the conditions

$$J^*_{g_0}(x_0) = q_{g_0} ,$$

$$x_i \leq U ,$$

$$x_0 = 0 .$$

After $N$ iterations, repeat the following step for $k = N, \ldots, 1$.

(d) Set $u^*_{g_k} = u^*_g(x^*_{g_k})$ where $x^*_{g_k} = \text{argmax}_{x_{g_k}} J^*_g(x_{g_k})$, and $x^*_j = x^*_j(x^*_{g_k})$ for every model $j$ connected to an input of model $g_k$.  

50
This algorithm is simply a generalization of the previous DP algorithm; if you apply this algorithm to a pipeline network, the computation and steps needed are identical to those in Algorithm 3.1. When the network is not a pipeline, this algorithm computes the optimal quality outputs of all of the models starting at the leaves, and moves towards the root, at each step choosing a new model whose inputs are connected to models that have been already considered. Thus, after the optimal output quality for each model has been determined, the maximum value of $J_{g_{N}}^{*}(g_{N})$ is the optimal value of the output quality for the network. The optimal set of controls is simply the collection of $u_{g_{k}}^{*}$'s that are computed by the algorithm.

For this algorithm, we have kept the restrictions $0 \leq \sum_{i=1}^{N} u_{i} \leq U$ and $u_{i} \geq 0$ that Zilberstein imposes. This is fine if all of the processors are to be implemented and run in series, but in a distributed environment, the processors generally run in parallel as soon as their inputs are ready. If we assume the same network structure, and that $u_{i}$ corresponds to the time allocated to processor $i$, the total time required by the network is $U_{N} = u_{N} + \max_{i}(U_{i})$ where $N$ is the root of the network and $U_{i}$ is the time required for the subtree rooted at processor $i$ defined similarly, where $i$ ranges over all processors connected to the input of processor $N$. Another possible criterion is that the total time between successive outputs of the network must be less the maximum time usage by any single processor while at the same time imposing a maximum latency like one of the previously mentioned restrictions. It is not difficult to modify the preceding algorithm to accommodate these new constraints; all that needs to be done is to modify step (b) such that the valid $x_{j}$ for each iteration satisfy a different criterion with respect to $x_{g_{k}}$ and $u_{g_{k}}$, namely

$$x_{g_{k}} = u_{g_{k}} + \max_{j} x_{j}.$$ 

The fact that there are other natural sets of restrictions on the control inputs other than a simple additive constraint means that any approach designed to optimize output quality of a network must be able to handle these varying types of restrictions. The models we have introduced so far allow for somewhat arbitrary restrictions on
the controls, but the algorithms that are used to optimize quality have assumed an additive restriction. Although the algorithms developed in the next chapter will also assume one of the two restrictions mentioned above for simplicity, they can be easily modified to work for any restriction that can be incorporated into this framework.
Chapter 4

More Complex Networks

In this chapter, we will begin to consider more complex networks of approximate processors, consisting of interconnections of processors with multiple outputs. We will concentrate on finding sets of control inputs that optimize the output quality of a network. We define a policy \( \pi \) as a member of \( \mathcal{U}_N \) (i.e., a feasible set of control inputs to the APN); an optimal policy \( \pi^* \) is a policy that maximizes the output quality of the network. The general problem that we wish to solve is the following.

**Problem 4.1 (Desirable Output Quality)** Given a network \( N \), determine the optimal policy \( \pi^* = (u_1^*, u_2^*, \ldots, u_N^*) \) such that

\[
y_N^* = Q(\pi^*, q) = \operatorname{opt}_{\pi \in \Pi} Q(\pi, q),
\]

where \( q \) is a vector of all quality inputs to \( N \), and \( Q(\cdot) \) is the function describing the quality outputs of \( N \) as a function of the control inputs and quality inputs.

The operator \( \operatorname{opt} \) optimizes the quality \( Q(\cdot) \) with respect to some set of criteria and finds the most desirable result. In the previous chapter, \( \max \) could be used instead of \( \operatorname{opt} \) because all models were required to have exactly one quality output, but in this chapter, when we look at multiple-output processors, we have to consider the question of how to choose the proper definition of \( \operatorname{opt} \). In order to deal with the problem of defining \( \operatorname{opt} \), we will look at a multidimensional version of the dynamic
programming algorithm, in which there are multiple competing criteria that must be maximized simultaneously.

4.1 Multidimensional Networks

Before we consider processors with multidimensional quality outputs, we will briefly discuss the issue of models with multidimensional control inputs. Up until now, we have focused on quality models that have single dimensional control inputs. Because these control inputs are used as the variables to optimize over, allowing multiple control inputs to a single model increases the complexity of the optimization steps required by the algorithms by increasing the number of possible control values, but does not fundamentally change the problems we wish to solve. Thus, we will always assume that processors have a single dimensional control input in order to simplify the discussion.

When we allow multiple quality outputs from the individual processors, we are interested in the problem of finding a set of control inputs that results in a “desirable” output quality. Consider a simple case in which we have a system that generates video images with two relevant measures of quality: resolution and bit depth. If there is a limited time allotment in which to operate the system, how does one choose between high resolution/few bits, low resolution/many bits and medium resolution/some bits? An example of this is shown in Figure 4-1. The three modified images ((b), (c), and (d)) show the effect of trading off resolution with the number of bits per pixel in an image, subject to the constraint that the total number of bits used to represent the entire image stays constant. The question of which image is most suitable to pass on for further processing arises from the fact that the quantity of interest, the output quality, is now a vector rather than a scalar. Depending on whether further processing of the image requires sharp spatial or color detail, or both, one of the three modified images will be most suitable. There have been many approaches to dealing with these types of problems (see [7] and [8]), but we will only concentrate on a few
Figure 4-1: Figure (a) shows the original image at 8 bits/pixel (bpp), with a resolution of $600 \times 400$ pixels. (b), (c), and (d) show the same image at 1 bpp and $600 \times 400$, 4 bpp and $300 \times 200$, and 8 bpp and $216 \times 144$, respectively. Which do you like best?
basic ideas here. Our approach will consider the problem of maximizing the output quality of an multidimensional APN as a multi-criteria optimization problem, i.e., we are trying to simultaneously optimize a number of different, competing criteria.

The following discussion will assume that the network of interest is a multidimensional pipeline, with (possibly) varying numbers of output quality measures at each stage but a single dimensional control input per processor.

4.1.1 Multi-criteria Dynamic Programming (MDP)

The main objective of many existing multi-criteria optimization methods is to reduce the number of possible solutions to a smaller set from which the user can select the optimum, based on the user's particular preferences. For instance, if we have the same video generation system just described and the particular user at a given time is known to prefer high resolution over many bits, then we may discard the unwanted alternatives and present the one with most utility for the user, e.g., select (b) in Figure 4-1. If we have enough information about the user, we may formulate a more concrete model of a user's preferences, via a utility function. A utility function is simply a mapping from a quality vector to a single measure of usefulness for the user. In this way, the multi-criteria optimization problem is reduced to a single dimensional maximization of the utility function. This is fine when the utility function and the overall network are fixed and known, but if the network is changing, or we do not have access to a specific utility function, this method fails to provide us with any useful answers. Even if the utility function is known, how can we adapt existing methods to use the utility function to deal with the complexity of the optimization problem? Dynamic programming requires an optimization to occur at each stage, but it has been shown in [9] that we cannot just maximize the utility function evaluated at the output of each stage in order to reduce the multidimensional problem back to a single dimensional dynamic programming problem\(^1\). Thus, we have to deal with a more

\(^1\)However, in some cases, notably when the performance profiles are sums or products of their arguments and the utility function is a linear combination or geometric combination, respectively,
general form of dynamic programming in order to optimize the performance of these networks.

In order to use DP to optimize these networks, we modify Problem 4.1 as follows:

**Problem 4.2 (Maximum (or Non-Dominated) Output Quality (Multiple Outputs))**

*Given an APN, determine the set \( \Pi^* \) such that for each \( \pi^* \in \Pi^* \):

\[
y^*_N = Q(\pi^*, q) < Q(\pi, q) \quad \forall \pi \neq \pi^*
\]

(4.2)

where \( q \) is a vector of all quality inputs, and \( Q(\cdot) \) is the function describing the unconnected outputs of the network as a function of the control inputs and initial input qualities.

In this case, \( a < b \) for vectors \( a \) and \( b \) if \( a \leq b \) and \( a \neq b \), and we say that \( a \) is dominated by \( b \) or that \( b \) dominates \( a \). The set of all \( Q(\pi^*, q) \) for every \( \pi^* \in \Pi^* \) is referred to as the non-dominated set. The set \( \Pi^* \) is referred to as the efficient set. We also define \( \pi^*(f) \) as the optimal policy \( \pi^*(f) \) with respect to a particular utility function \( f(\cdot) \), which is the policy that attains the output quality whose utility value is the greatest, i.e.,

\[
\pi^*(f) = \arg\max_{\pi \in \Pi^*} f(Q(\pi, q)).
\]

(4.3)

Assuming we can compute the non-dominated set, the question remains as to how we can use it to generate a policy that will optimize the output quality of the APN with respect to some utility function or set of criteria. The following theorem can be found in [7], and is restated and proved here in order to establish the usefulness of the non-dominated set.

**Theorem 4.1** The above definition of the non-dominated set guarantees that any optimal policy with respect to any strictly monotonic utility function will be a member of \( \Pi^* \).

**Proof:** Assume that \( \pi_0 \) is an optimal policy with respect to a strictly monotonic utility function \( f(\cdot) \), i.e., \( f(Q(\pi_0, q)) = \max_{\pi} f(Q(\pi, q)) \). Also assume that \( \pi_0 \notin \Pi^* \). Thus, we can actually solve the multidimensional pipeline problem by maximizing the utility function at each stage.
there exists a $\pi_1$ such that $Q(\pi_1, q) > Q(\pi_0, q)$ and because $f(\cdot)$ is strictly monotonic, $f(Q(\pi_1, q)) > f(Q(\pi_0, q))$ which contradicts the assumption that $\pi_0$ is an optimal policy.

Thus, if the non-dominated set can be computed, then maximizing a particular strictly monotonic utility function is simply a matter of comparing the values of the utility function evaluated at each member of the non-dominated set in order to find a maximum. In the absence of knowledge of the utility function beforehand, Problem 4.2 provides a method of computing all viable policies that could possibly be optimal.

With this new setup, we can describe a multidimensional DP algorithm (adapted from [9]) that will determine the non-dominated set for a given pipeline network.

**Algorithm 4.1 (Maximum (or Non-Dominated) Output Quality — MDP)** Solve the following equations

\[
J_i^*(x_i) = \text{ndom} \{Q_i(u_i, J_{i-1}^*(x_i - u_i)) \mid 0 \leq u_i \leq x_i, J_{i-1}^*(\cdot) \in J_{i-1}^*(\cdot)\},
\]

\[
x_{i-1} = x_i - u_i,
\]

starting at $i = 1$ and continuing through $i = N$, subject to the conditions

\[
J_0^*(x_0) = q_1,
\]

\[
x_i \leq U,
\]

\[
x_0 = 0.
\]

The expression "ndom $A$" returns all of the vectors in the set $A$ that are not dominated by any other vector in the set. The basic idea behind the algorithm is that the non-dominated output quality vectors of a given model $i$ (set $J_i^*(\cdot)$) must be generated by passing a non-dominated output quality vector (member of $J_{i-1}^*(\cdot)$) from model $i - 1$ to the input of model $i$. This is true because the monotonicity of the performance profiles guarantees that if $J_{i-1}^*(\cdot)$ dominates $J_{i-1}'(\cdot)$, then $Q_i(u_i, J_{i-1}(\cdot))$ dominates $Q_i(u_i, J_{i-1}'(\cdot))$. So, at each stage of the algorithm, all that needs to be done is to
generate the new set of non-dominated quality vectors based on the previous set. At the end of the algorithm, $\text{ndom}\{\mathbf{J}_N^*(x_N) \mid \forall x_N\}$ is the non-dominated set.

This algorithm is simply a multidimensional version of Algorithm 3.1, which calculated the maximum output quality for a pipeline with a single dimension of input and output quality for each processor. Whereas Algorithm 3.1 is functionally equivalent to Zilberstein's method of local compilation, this new algorithm can be used as a method of compiling anytime algorithms that have multiple dimensions of quality. Note the flexibility in this new algorithm with respect to the number and type of quality inputs and outputs that are present in the pipeline. As long as the number of output quality measures of some processor $i$ is equal to the number of input quality measures of processor $i + 1$, this algorithm will produce the non-dominated set of output quality vectors. The non-dominated set may, in general, consist of an infinite number of vectors, depending on the particular performance profiles and control inputs. This does not cause problems with the statement of the algorithm, but does cause difficulty with the implementation. In order to circumvent this problem in a practical situation, we can assume that the control inputs to an APN can take only one of a finite number of values. This ensures that the number of vectors in the non-dominated set will also be finite. This assumption is made in the next section in the development of distributed quality optimization algorithms.

Although this MDP algorithm only calculates the non-dominated set for pipeline networks, it can be adapted to be used with multidimensional forward tree networks using a variation of Algorithm 3.2. However, this variation will not be presented here as it is included as a restricted case of the final algorithm in Section 4.2.4. Instead, we will now focus our attention on calculating the non-dominated set for physically distributed networks.
4.2 Distributed Local Compilation

In this section we will describe distributed algorithms for generating the non-dominated set without a centralized computing agent. The algorithms will be developed for one dimensional pipelines as well as pipelines with multiple dimensions of quality, and for forward tree networks. The final algorithm provides a distributed method of computing the non-dominated set of network output quality values for a multidimensional forward tree network. This can, therefore, handle any of the other types of networks discussed, and can be restricted to compute the non-dominated set in a manner equivalent to that of the centralized algorithms discussed up to this point.

For now we will concentrate only on processors for which the allowable control input is one of a finite set of values, e.g., \{0, 1, 2, 3\}. With this restriction, each processor need only consider a finite number of possible input qualities in order to optimize its own output quality, simplifying the problem of communicating the results to other processors.

The previous MDP algorithm allows the non-dominated set of output quality vectors to be computed using knowledge about all of the processors in a network and their characteristics. In a real situation, however, the individual processors may not have much knowledge, or may not have instant knowledge, about the other processors in the network. In this case, we must come up with a distributed algorithm such that each processor can keep working with the knowledge it has until it receives new information. In [10], a general distributed dynamic programming algorithm is presented which accomplishes this goal, and it is shown that the algorithm converges to the same solution as the traditional dynamic programming algorithm under some weak restrictions. In fact, because we are restricting our networks to be topologically equivalent to trees (at the most complex), we do not need the full generality of the algorithm presented in [10], and proving that our distributed algorithms below will find the non-dominated set of output quality values within a bounded time is not difficult. We will assume that each model in a network has a means of communicating with each of the other models in the network with which it is connected. In the
following discussion, a single dimensional pipeline refers to a pipeline network in which each processor has a single dimensional quality input and quality output. A multidimensional pipeline refers to a pipeline network in which at least one processor has a quality input or output that is multidimensional.

4.2.1 Single Dimensional Pipeline

We now describe an algorithm that can be used to compute the maximum output quality and corresponding optimal policy that achieves the maximum output quality of a single dimensional pipeline. This algorithm is essentially a method of dividing the computations required for Algorithm 3.1 such that each model can perform a set of computations independently of the other models, using only information available from its neighbors in the pipeline. The distributed method of computation used in this algorithm benefits the flexibility and adaptivity of the network in a few ways. First, the number and type of processors that are present in the network need not be known in order for the algorithm to succeed. This means that the actual computation required can be designed into the physical processor, which can then be used in a variety of different networks, as the processor does not have to be reconfigured when it is placed in a new network. Second, because the computation may be performed repeatedly throughout the entire operating time of the network, any changes in the network, either parametric changes in a single processor, or replacements, additions or deletions of processors in the network, will eventually be incorporated into the quality measures and thus the algorithm will achieve success by adapting to the changing environment.

Because the computation and communication required for this algorithm is distributed across all of the processors in a network, the algorithm is best described from the point of view of the individual models. The statement of the algorithm takes the form of a description of the various activities that the individual models must complete in order for the algorithm to succeed. The basic idea behind the algorithm is that each model need only concentrate on computing the optimal quality output that
it can manage for each possible quality input from the preceding model. Then, the model must transmit these optimal output qualities to the next model so that that model may perform a similar computation. After some amount of time has passed, the final model in the pipeline must be able to calculate the optimal quality output of the entire pipeline, as well as the optimal policy so that all of the models will know the correct control inputs needed to achieve the best quality. The statement of the algorithm below is followed by a proof that the quality output and policy computed by the final model will be optimal after a bounded time, under an assumption on the time required for the computation that each processor must accomplish. As before, we define $x_i$, the cumulative control input of the models from 1 through $i$, as the sum of the individual control inputs, $\sum_{j=1}^{i} u_j$. Again, this is not the only possible choice for $x_i$, but is consistent with our earlier algorithms.

Each model has four possible states of execution: Compute, Transmit, Update, or Idle. When in the Compute state, model $i$ computes estimates of its possible optimal quality outputs; when in the Transmit state, model $i$ transmits its current estimate of its optimal quality output values to model $i + 1$ (unless $i = N$). The model transmits information about the optimal policy to the previous processor when in the Update state. When in the Idle state, a model does no computation or transmission. We denote the current estimate of the optimal quality outputs of a model $i$ by $\hat{J}_i(x_i)$ where $x_i$ is the cumulative control input given to the preceding models (e.g., the total time, power, cost, etc.). The value of $x_i$ is important not only in the computation of the optimal output quality, but also in the generation of the optimal policy, because the set of $x_i$ are used to find the optimal policy in the Update state. Each model must store current estimates for each value of $x_i$, along with the value of the control input $\hat{u}_i(x_i)$ that achieves the current estimate for that value of $x_i$, and a current state $\hat{x}_i$ that the model uses to select the actual control input used. The values of $\hat{x}_i$ and $\hat{u}_i(\hat{x}_i)$ are not used in the quality optimization computation, but is used by the actual processor (not the model) to control the real processing done by the processor. We will use $J^*_i(x_i)$ to denote the optimal quality outputs of
a model $i$, the same definition that was used for Algorithm 3.1. From the previous centralized algorithms, we know that as soon as $\hat{J}_k(x_k) = J^*_k(x_k)$ for $k = 1, 2, \ldots, i$, we can compute $J_{i+1}(x_{i+1})$ easily. This idea will be used to prove the convergence of the following algorithm. In order to compute the maximal output quality, each model (denoted by $i$) need only perform the following tasks in the COMPUTE, TRANSMIT, and UPDATE states.

**COMPUTE:** Compute the new estimate

$$\hat{J}_i(x_i) = \max_{0 \leq u_i \leq x_i} Q_i(u_i, \hat{J}_{i-1}(x_i - u_i)),$$  \hspace{1cm} (4.4)

for each possible value of $x_i$. Store this and the corresponding maximizing $\hat{u}_i(x_i)$.

**TRANSMIT:** Transmit $\hat{J}_i(x_i)$ from model $i$ to model $i+1$ (unless $i = N$) for each value of $x_i$.

**UPDATE:** Set control input to $\hat{u}_i(\hat{x}_i)$, and transmit $\hat{x}_{i-1} = \hat{x}_i - \hat{u}_i(\hat{x}_i)$ to model $i-1$ as its new current state (unless $i = 1$).

The selection of initial conditions for some of the quantities above (e.g., $\hat{J}_{i-1}(\cdot), \hat{x}_i$) is not important as their values will only affect the values computed by the algorithm in the first few stages. However, in order to make the algorithm well-defined, we do have to impose the following conditions

$$\hat{J}_0(x_0) = J^*_0(x_0) = q_1.$$

There is one special case, that of the final model in the network, model $N$, which has an additional task it must perform immediately before the **UPDATE** task:

**PRE-UPDATE:** Calculate $\hat{x}_N = \arg\max_{x_N} \hat{J}_N(x_N)$.

The only restriction we will place on the timeliness and order in which the models must perform these tasks is that there must exist a positive scalar $P$ such that, for
every processor $i$, every time interval of length $P$ contains at least one computation for $i$, at least one transmission interval from $i$ to node $i+1$, for $1 \leq i \leq N - 1$, and at least one update from $i$ to node $i - 1$, for $2 \leq i \leq N$.

**Convergence Time** With these assumptions, we can bound the time that it takes for this approach to compute the maximal output quality at the last processor and the optimal policy that achieves this level of output quality. The above assumption on the time required to accomplish the necessary tasks allows us to state and prove the following theorem.

**Theorem 4.2** Given an APN pipeline with $n$ processors, each of which performs all of its tasks (Compute, Transmit, and Update) at least once within every time interval of length $P$, the time needed for the estimates of the optimal output quality and of the optimal policy to converge to their true values is less than $3nP$.

**Proof:** During the interval $(0, P]$, model 1 computes its exact optimal output quality values $J^*_1(x_1)$ for each value of $x_1$, because 1 is the first model in the pipeline, and thus has no dependence on any other estimates. During the interval $(P, 2P]$, 1 transmits these optimal values to 2. Now, if model $i-1$ has transmitted to model $i$ the true optimal output quality values $J^*_i(x_{i-1})$ (not estimates) at time $2(i-1)P$ for each possible $x_{i-1}$, then at time $2(i-1)P + P$ model $i$ will have computed the true optimal output quality values $J^*_i(x_i)$ because $i$ only depends on values of $J^*_{i-1}(x_{i-1})$. And, at time $2iP$, $i$ will have transmitted these values to $i + 1$. Thus, by induction, by time $2nP$ the optimal output quality values $J^*_N(x_N)$ will have been computed at all states.

Then, we can follow similar reasoning to show that at time $2nP + (n - i)P$, $i$ will have received its optimal control input from $i + 1$, and thus by time $3nP$ the entire optimal policy will have been computed and stored in the processors.

This bound on the convergence time of the algorithm allows one to make predictions about the time needed to recalculate the optimal policy after changes take place in the network, and choose the size of the set $\mathcal{U}$ for each processor, depending on the speed with which it can calculate new value function estimates. For example, if it is known that a particular pipeline will always contain fewer than 10 processors, and
that each processor can compute, transmit and update every 25ms, this approach to computing the optimal policy will work best when the time between appreciable changes in the pipeline (parametric or structural variation) is greater than 750ms.

It is useful to note that in the case of the pipeline, if we simply tell each processor to wait until it receives the transmission of the optimal output quality function \( J^*(\cdot) \) from the previous processor before doing any computation or transmission, we would be exactly implementing Algorithm 3.1. However, the more general distributed approach allows the network to adapt to changes in the structure as well as deal with unknown network structures over time.

Remember that we assumed earlier in this chapter that the control inputs to all of the models took values in some finite set. This implies that \( x_i \) also takes values in some finite set. We can now understand more clearly what the models are doing in each of the states described above. In its COMPUTE state, model \( i \) steps through each possible value of \( x_i \), and finds the combination of \( u_i \) and \( x_{i-1} \) such that \( x_i - u_i = x_{i-1} \) and that \( Q_i(u_i, \hat{J}_{i-1}(x_{i-1})) \) is maximized. The computation required is then simply an iteration through the possible pairs of values of \( x_i \) and \( x_{i-1} \). Because both \( x_i \) and \( x_{i-1} \) are members of some known finite set, this sequence of iterations will complete after some finite bounded time \( C_i \) has elapsed (assuming the evaluation of \( Q_i(\cdot) \) cannot take unbounded time). In the TRANSMIT state, model \( i \) sends each of the values \( \hat{J}_i(x_i) \) that it computed in its last COMPUTE state to model \( i + 1 \). Because there are a finite number of possible values of \( x_i \), and assuming that the transmission of a single value does not take unbounded time, the time required by this process will also be bounded by some finite number \( T_i \). Finally, in the UPDATE state, model \( i \) makes a single computation and sends a single piece of data to model \( i - 1 \), and for model \( N \), the PRE-UPDATE task also takes bounded time. So the time required for the UPDATE task is also bounded by some finite number \( R_i \). So, if we define \( P = \max_i C_i + \max_j T_j + \max_k R_k \), and require the models to perform the three states repeatedly in the order COMPUTE, TRANSMIT, then UPDATE, our assumption on the time required for each processor to perform its tasks is satisfied. The only additional
restrictions that are needed are bounded time assumptions on the time required to evaluate the performance profile and on the time required to transmit a piece of data between two processors, both of which are very easily satisfied in many practical situations. This (or similar) reasoning can also be used to show the practicality of the bound \( P \) for the following algorithms as well.

4.2.2 Single Dimensional Trees

It should also be clear that the above distributed DP algorithm can be extended to work for any arbitrary forward tree network in the same way as the centralized DP algorithm was extended in the previous chapter. Each processor will now have as neighbors processors that are connected to one of its inputs. The only difference is the higher complexity in the computation required by each model, as the multiple inputs will increase the number of input combinations considered by each model. Each model will have to consider all possible combinations of input qualities from the input processors. Also, each processor will now have a slightly more complicated rule for computing \( x_i \), and will require more storage in order to keep enough information about the preceding processors to generate an optimal policy after the optimal output quality has been found.

In the case where the maximum allowed processing time of any path from an input to an output of the network is restricted, the model must optimize its output quality using the maximum of the cumulative time usage for each of the input models, e.g., for a model \( i \) which has inputs from processors \( j \) and \( k \):

\[
\hat{J}_i(x_i) = \max_{0 \leq u_i \leq x_i} Q_i(u_i, \hat{J}_j(x^j_i - u_i), \hat{J}_k(x^k_i - u_i)) ,
\]

(4.5)

where \( x_i = \max(x^j_i, x^k_i) \). If we know that all of the performance profiles are monotonic in their control inputs, we must have that \( x^j_i \) is always equal to \( x^k_i \). Otherwise, an increase in the smaller of the two (which does not increase \( x_i \)) would cause an increase

66
in quality. Then, we could simplify the previous equation to

\[ \hat{J}_i(x) = \max_{0 \leq u_i \leq x_i} Q_i(u, \hat{J}_j(x_i - u_i), \hat{J}_k(x_i - u_i)) . \]  \hspace{1cm} (4.6)

However, if we restrict the maximum total time usage by all processors as we have been doing, where the total resource usage is simply the sum of the individual resource requirements for each model, each processor must optimize using the sum of the allocations propagated forward by the input processors. In other words, for the same two-input processor above, we have

\[ \hat{J}_i(x_i) = \max_{0 \leq u_i \leq x_i} Q_i(u_i, \hat{J}_j(x_j), \hat{J}_k(x_k)) , \]  \hspace{1cm} (4.7)

where \( x_i = u_i + x_j + x_k \). Now, the optimization must be done over two variables, \( u_i \) and either \( x_j \) or \( x_k \).

The computation needed in each state for the models is similar to what was needed for a pipeline network. The main difference is the need to store multiple values that allow the processor to send current state estimates to the multiple preceding processors in the UPDATE state, and increased computation required for the output quality estimates in the COMPUTE state. Also, the method of calculating the control input from the current state of the current processor and the current state of the preceding processors is more complicated.

**COMPUTE:** Compute the new estimate

\[ \hat{J}_i(x_i) = \max_{0 \leq u_i \leq x_i, \{\hat{x}_{j_k}\}} Q_i(u_i, \hat{J}_{j_1}(x_{j_1}), \hat{J}_{j_2}(x_{j_2}), \ldots) , \]  \hspace{1cm} (4.8)

for each state \( x_i \) where the \( j_k \) indicate models whose outputs are connected to the input of model \( i \). Store these estimates and the corresponding maximizing values \( \hat{x}_{j_k} \) which attain these estimates.

**TRANSMIT:** Transmit \( \hat{J}_i(x_i) \) from model \( i \) to the model (if any) connected to the quality output of \( i \) for each value of \( x_i \).
**UPDATE:** Compute control input \( \hat{u}_i = \hat{x}_i - \sum_{j_k} \hat{x}_{j_k} \) and transmit \( \hat{x}_{j_k} \) to model \( j_k \) as its new current state.

**Convergence Time** We can use a similar argument to what was used in determining the time for a pipeline to converge to the optimal quality output values, but now to show that the maximum time needed to calculate the optimal quality output values for a forward tree network is \( 3mP \), where \( m \) is the number of segments in the longest path in the tree. This result illustrates the usefulness of the distributed DP algorithm, as a tree network that is topologically a complete binary tree of length \( m \) has \( 2^m - 1 \) processors. The amount of time that is needed by a centralized DP algorithm would be exponential in the height of the tree, rather than linear.

The following example illustrates the ideas from the previous algorithm with a concrete example, but also shows how the algorithm can be modified to accommodate a different constraint on the time required by the processors.

**Example (Distributed DP for an Example APN)** Consider a simple example APN, with two single-dimensional quality models, each connected to an input of a quality model with a one-dimensional quality output, but a two-dimensional input. The network is depicted in Figure 4-2. We will assume that the controls \( u_1 \), \( u_2 \) and \( u_3 \) must satisfy the inequality \( 0 \leq \max(u_1, u_2) + u_3 \leq 3 \). The performance profiles \( Q_1(\cdot) \) and \( Q_2(\cdot) \) are summarized in Figure 4-3. The control inputs \( u_1 \), \( u_2 \) and \( u_3 \) all take values in \( \{0, 1, 2, 3\} \). From these tables, we can see that \( \mathcal{Y}_1 = \{0.0, 0.2, 0.4, 0.6, 0.8, 0.9, 1.0\} \) and that \( \mathcal{Y}_2 = \{0.0, 0.1, 0.2, 0.4, 0.5, 0.7, 0.8, 0.9, 1.0\} \).

Also, assume
\[
Q_3(u_3, y_1, y_2) = \frac{y_1 + y_2}{2} \left(1 - e^{-2u_3}\right).
\]

Given this information, we can see how the distributed DP algorithm would work. Assume that \( q_1 = q_2 = 1.0 \). In the first interval, \([0, P)\), processor 1 will set
\[
\hat{J}_1(0) = 0.6 \quad \hat{J}_1(1) = 0.8 \quad \hat{J}_1(2) = 0.9 \quad \hat{J}_1(3) = 1.0
\]
Figure 4-2: An example of a forward tree network.

and processor 2 will set

\[
\hat{J}_2(0) = 0.5 \; , \; \hat{J}_2(1) = 0.8 \; , \; \hat{J}_2(2) = 0.9 \; , \; \hat{J}_2(3) = 1.0 .
\]

Then, in the next interval, \([P, 2P]\), processors 1 and 2 will transmit these values to processor 3, which must then calculate the optimal output qualities \(y_3\) subject to the given restriction on the control inputs. For the case when we use all available resources, \(x_3 = 3\) and thus,

\[
\hat{J}_3(3) = \max_{u_3 = 3 - \max(x_1, x_2)} Q_3(u_3, \hat{J}_1(x_1), \hat{J}_2(x_2)) ,
\]

which takes its maximum value when \(x_1 = x_2\) because of the monotonicity of the perfor-

<table>
<thead>
<tr>
<th>(Q_1(\cdot))</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (q_1)</td>
<td>0.0</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
</tr>
<tr>
<td>0.25, 0.5</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
<td>0.8</td>
</tr>
<tr>
<td>0.5, 0.75</td>
<td>0.4</td>
<td>0.6</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>0.75, 1.0</td>
<td>0.6</td>
<td>0.8</td>
<td>0.9</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(Q_2(\cdot))</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (q_2)</td>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>0.25, 0.5</td>
<td>0.1</td>
<td>0.2</td>
<td>0.5</td>
<td>0.8</td>
</tr>
<tr>
<td>0.5, 0.75</td>
<td>0.4</td>
<td>0.7</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>0.75, 1.0</td>
<td>0.5</td>
<td>0.8</td>
<td>0.9</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Figure 4-3: Performance profiles \(Q_1(\cdot)\) and \(Q_2(\cdot)\). Values in the tables denote value of the quality output.

69
mance profiles, so

\[ \hat{J}_3(3) = \max_{u_3} Q_3(u_3, \hat{J}_1(3 - u_3), \hat{J}_2(3 - u_3)) . \]

Now we simply have to evaluate each of the four possibilities, using the performance profile \( Q_3(\cdot) \):

\[
\begin{align*}
Q_3(0, \hat{J}_1(3), \hat{J}_2(3)) &= Q_3(0, 1.0, 1.0) = \frac{1}{2} (1 - e^0) = 0 , \\
Q_3(1, \hat{J}_1(2), \hat{J}_2(2)) &= Q_3(1, 0.9, 0.9) = \frac{0.9}{2} (1 - e^{-2}) = 0.3891 , \\
Q_3(2, \hat{J}_1(1), \hat{J}_2(1)) &= Q_3(2, 0.8, 0.8) = \frac{0.8}{2} (1 - e^{-4}) = 0.3927 , \\
Q_3(3, \hat{J}_1(0), \hat{J}_2(0)) &= Q_3(3, 0.6, 0.5) = \frac{0.55}{2} (1 - e^{-6}) = 0.2743 .
\end{align*}
\]

Therefore, the optimal policy for this particular network, when the input qualities \( q_1 \) and \( q_2 \) are equal to 1.0, is to set \( u_3 = 2 \) and \( u_1 = u_2 = 1 \). However, when the input qualities are \( q_1 = q_2 = 0.1 \), we have the following:

\[
\begin{align*}
\hat{J}_1(0) &= 0.0 , \quad \hat{J}_1(1) = 0.2 , \quad \hat{J}_1(2) = 0.4 , \quad \hat{J}_1(3) = 0.6 , \\
\hat{J}_2(0) &= 0.0 , \quad \hat{J}_2(1) = 0.1 , \quad \hat{J}_2(2) = 0.2 , \quad \hat{J}_2(3) = 0.5 ,
\end{align*}
\]

and

\[
\begin{align*}
Q_3(0, \hat{J}_1(3), \hat{J}_2(3)) &= Q_3(0, 0.6, 0.5) = \frac{0.55}{2} (1 - e^0) = 0 , \\
Q_3(1, \hat{J}_1(2), \hat{J}_2(2)) &= Q_3(1, 0.4, 0.2) = \frac{0.3}{2} (1 - e^{-2}) = 0.1297 , \\
Q_3(2, \hat{J}_1(1), \hat{J}_2(1)) &= Q_3(2, 0.2, 0.1) = \frac{0.15}{2} (1 - e^{-4}) = 0.0736 , \\
Q_3(3, \hat{J}_1(0), \hat{J}_2(0)) &= Q_3(3, 0.0, 0.0) = \frac{0}{2} (1 - e^{-6}) = 0 .
\end{align*}
\]

Now, \( \hat{J}_3(0) = 0.1297 \) when \( u_3 = 1 \) and \( u_1 = u_2 = 2 \). In addition to illustrating the use of the algorithms we have developed so far, this example has a few other interesting points. First, it incorporates an alternative restriction on the control inputs to the system; the possible control inputs are limited to be those such that the largest sum of the controls along any
path from an input to an output is restricted to be less than 3. This is the same restriction mentioned at the end of Chapter 3 as an alternative to Zilberstein’s additive constraint. Second, it provides an example of how the values of the quality inputs to the network can affect the selection of the optimal policy. In the examples given previously, the quality inputs were such that they could not affect the choice of the optimal policy, only the value of the maximal output quality.

4.2.3 Multidimensional Pipelines

Now we look at applying the distributed DP framework to a multidimensional pipeline, i.e., a pipeline with multiple dimensions of quality at the inputs and outputs of each stage. To use the previous distributed DP techniques to calculate a vector-valued optimal value function, we must combine our previous distributed DP-based algorithm with the MDP algorithm.

We may think of the computation each model must do in the following way: each model $i$ must determine the set of all the best (non-dominated) output quality vectors it can produce, given some set of input quality vectors. It must calculate this set for each possible value of $x_i$. If the preceding model(s) have already calculated their set(s) of non-dominated output quality vectors, model $i$ need only consider each possible collection of these vectors (one from each of the preceding models) as the input quality vector $q_i$, and the corresponding resource usage $x_j$ for the particular input model $j$, in order to find the optimal output quality vectors. The difference between the centralized and distributed situations is that the processors in a distributed setting perform this computation before they are guaranteed that the preceding processors have actually all completed their computation. So, to make sure that the non-dominated set of output qualities is computed at the final output model, each model repeats the computation as long as the sets of non-dominated output quality vectors it is producing are changing between computation iterations.

The differences between the distributed DP algorithm for a multidimensional pipeline and for a single dimensional pipeline are the processing done in the Com-
PUTE phase, and the data transmitted in the TRANSMIT and UPDATE phases. The new descriptions of these phases are given here. In order to generate the optimal policy in the UPDATE state, we now need more information to recover the state of the preceding processor. We need both the value of $\hat{x}_{i-1}$, and the quality vector $\hat{J}_i(\hat{x}_{i-1})$ that is associated with it because the non-dominated set now has many elements for each value of $\hat{x}_{i-1}$, instead of just one element as in a single dimensional network.

**COMPUTE:** Compute the new estimate

$$\hat{J}_i(x_i) = \text{nondom} \left\{ Q_i(u_i, \hat{J}_{i-1}(x_i - u_i)) \ | \ 0 \leq u_i \leq x_i, \hat{J}_{i-1}(\cdot) \in \hat{J}_{i-1}(\cdot) \right\} , \quad (4.9)$$

for each state $x_i$ and set it as the new estimate $\hat{J}_i(x_i)$ after all computation is complete. Store each member of the estimate set along with the corresponding maximizing $\hat{u}_i(x_i)$.

**TRANSMIT:** Transmit $\hat{J}_i(x_i)$ from model $i$ to some $i+1$ (unless $i = N$) for each value of $x_i$.

**PRE-UPDATE:** Calculate $\hat{x}_N = \arg\max_{x_N} \{ f(\hat{J}_N(x_N)) \ | \hat{J}_N(\cdot) \in \hat{J}_N(\cdot) \}$ and the corresponding $\hat{J}_N(x_N)$ that reaches the maximum utility if there is a utility function $f(\cdot)$ specified.

**UPDATE:** Set control input to $\hat{u}_i(\hat{x}_i)$, and transmit $\hat{x}_{i-1} = \hat{x}_i - \hat{u}_i(\hat{x}_i)$ and the $\hat{J}_{i-1}(\hat{x}_{i-1})$ that was used to compute $\hat{J}_i(\hat{x}_i)$ to model $i-1$ as its new current state if there is a utility function specified.

The **PRE-UPDATE** and **UPDATE** tasks are only performed if a utility function has been specified in order to choose between the different possible non-dominated output quality vectors. If there is no utility function specified, there is no way of determining an optimal policy, and so we can only hope to generate the non-dominated set.

The same convergence results hold in this case as in the single dimensional case. However, although the assumption that there exists some fixed processing interval $P$ is possible to apply, it may not be as easily determined as for the single dimensional

72
pipeline. In some extreme cases, computation of the non-dominated set can be extremely costly and have a size on the order of the number of all possible policies. Under the assumption that we have a finite number of processors in our network, each of which has a finite number of controls, we can at least say that some finite value for $P$ exists, and thus the algorithm will converge. For many networks, the size of the non-dominated set is more manageable, and will lead to a useful value of $P$.

It was noted above that the distributed DP algorithm corresponds to the traditional DP algorithm for a pipeline network if the processors performed their tasks in a particular fashion. The same is true in this case; if we again tell each processor to wait until it receives an update transmission from the previous processor before doing any computation or transmission, we would be exactly implementing the MDP algorithm, Algorithm 4.1.

### 4.2.4 Multidimensional Trees

We may also consider adapting the preceding algorithms to use with multidimensional forward tree networks. The adaptation is straightforward from the multidimensional pipeline and single dimensional tree algorithms given above, so only the description of the various model states will be given, with the same notation used in previous descriptions.

**COMPUTE:** Compute the new estimate

$$\hat{J}_i(x_i) = \text{ndom} \{Q_i(u_i, \hat{J}_{j_1}(x_{j_1}), \hat{J}_{j_2}(x_{j_2}), \ldots) \mid 0 \leq u_i \leq x_i, \hat{J}_{j_k}(\cdot) \in \hat{J}_{j_k}(\cdot)\},$$

(4.10)

for each state $x_i$ where $j_k$ indicate models whose outputs are connected to the input of model $i$. Store each member of the estimate set along with the corresponding set of values $\hat{x}_{j_k}$ that attain the estimate value.

**TRANSMIT:** Transmit $\hat{J}_i(x_i)$ from model $i$ to some $i+1$ (unless $i = N$) for each value of $x_i$. 

73
UPDATE: Compute current control input $\hat{u}_i = \hat{x}_i - \sum_{j_k} \hat{x}_{jk}$, and transmit $x_{jk}$ and the corresponding $\hat{J}_{jk}(x_{jk})$ that was used to compute $\hat{J}_i(x_i)$ to model $j_k$ as its new current state only if there is a utility function specified.

The same PRE-UPDATE task is performed here as was in the previous algorithm, and again the PRE-UPDATE and UPDATE tasks are only to be performed if a utility function has been specified.

This algorithm is the final step in the work presented in this thesis. We have moved from Zilberstein's networks, which were single dimensional with additively constrained control inputs and centralized optimization (compilation) algorithms to a situation in which we can deal with networks made up of processors with varying numbers of relevant quality measures, more possible types of restrictions on the control inputs with distributed versions of the original optimization algorithms. This allows us to finally apply the techniques put forward in the IR and anytime algorithm literature to systems with some of the attributes described in Chapter 1.

This last algorithm also reduces to the previous algorithms under certain circumstances. For instance, if the network of interest is actually single dimensional, the ndom operation may be replaced with a simple maximization and the resulting algorithm would be the distributed single dimensional tree algorithm in Section 4.2.2. Then, if we restrict the order in which the processing is to be done such that the processors do their computation in the sequence dictated by the iterations in Algorithm 3.2, we would reduce the algorithm to a non-distributed algorithm. Further, if the network is known to be simply a single dimensional pipeline, we can reduce the algorithm all the way to the original DP algorithm, Algorithm 3.1 first introduced in Chapter 3. In this way, we have extended existing results to include a larger domain of possible approximate processor networks, including ones with multiple quality measures, and ones that are physically distributed.
Chapter 5

Conclusion

We have presented a new framework for quality optimization in networks containing approximate processors. The framework incorporates previously studied types of networks and processor models as well as more complex networks and models. However, the area of distributed approximate processing is still relatively new, and there are many issues that remain to be dealt with. The following paragraphs describe a number of research topics that could be investigated as extensions of this work, both practical and theoretical.

Stochastic Descriptions  Many algorithms exhibit stochastic behavior This is true of certain control algorithms, and for some search methods. Eventually we will need a method of characterizing such algorithms using stochastic models, and will need to investigate problems associated with the models. Zilberstein did incorporate stochastic performance profiles in [4], but we have not included those ideas in our work so far. There is a large body of research in the area of stochastic DP algorithms that may be useful in helping to develop algorithms to deal with stochastic models. The development of these types of models and algorithms would prove very useful in practical situations where often the input data can only be described stochastically beforehand.
**Systems with Memory**  Another possible extension of the class of models presented here is to incorporate the ability for the quality outputs of a system to depend on previous as well as current quality inputs to the system. This would allow the development of finer-grained descriptions of processors, and therefore more accurate modeling. Areas of research that may prove useful in this investigation would be state-space modeling of systems, and related areas of control theory.

**More Complex Networks**  Although the algorithms described in this thesis are applicable for a useful class of networks, there are still a lot of other possible network configurations that are possible. The main weakness of the current methods is that they cannot handle networks that contain processors that send output data to many other processors. Zilberstein presents results showing that his local compilation methods are not applicable to such situations, as they would not produce optimal solutions, and that the problem of optimizing output quality for general networks is an NP-complete problem. For small-scale networks, however, the computation needed would not be too great, and methods to optimize output quality for these more complex networks would prove useful in practical situations.

**Other Algorithms / Objectives**  The algorithms we have developed here are strongly based on various DP algorithms; there may be other suitable algorithms that may be used for optimization problems similar to the ones addressed here. Also, there are a wide variety of other possible optimization problems that could arise in practice. For example, one situation may call for a constant output quality while minimizing the resource usage of the network. Control algorithms that induce other types of useful behavior such as robustness to model errors, quality variation, slowly changing control inputs, etc. are also useful in certain contexts. We have already begun to investigate a method for allowing the optimal policy to change continuously in response to small variations in various model parameters using a policy iteration technique. Another approach that may prove useful in reducing the computation needed when working with multidimensional networks is *generalized dynamic programming*. 
(GDP) in [9]. GDP establishes methods of using information about particular utility functions and performance profile structures to reduce the amount of computation required to compute the non-dominated set. It essentially eliminates alternatives at early stages that could not possibly lead to a non-dominated quality output.

Models for Real Systems There is always a need for more refined quality models of existing systems, or of newly developed systems. These help to both validate the model and show the range of applicability, and provide more examples to investigate and integrate into existing systems.

Simulation Environment Software implementations of most of the algorithms in this thesis have been written using various software tools, including MATLAB and Tcl, in an effort to test the algorithms. A more comprehensive software testbed for the approximate processor network framework would help to speed the validation of new algorithms and models by providing a consistent, standard simulation platform. We are considering a few different packages with which to build such a testbed, including Ptolemy (see [11]) and MATLAB's SIMULINK package. There has not been significant progress yet, but development in this area would be very beneficial to further work.
Bibliography


THESIS PROCESSING SLIP

FIXED FIELD: ill. ____________ name ______

index ________ biblio ________

➤ COPIES: Archives Aero Dewey Eng Hum
Lindgren Music Rotch Science

TITLE VARIES: □

NAME VARIES: □ Joelson

IMPRINT: (COPYRIGHT)

➤ COLLATION: 79p

➤ ADD. DEGREE: ________ ➤ DEPT.: ________

SUPERVISORS: ____________________________

NOTES: ____________________________

cat' r: ____________________________

➤ DEPT: E.E. ➤ YEAR: 1996 ➤ DEGREE: M.S.
➤ NAME: SECOR, Matthew J.