Centro de Investigación y de Estudios Avanzados del
Instituto Politécnico Nacional
Departamento de Ingeniería Eléctrica
Sección de Computación

Hardware Level Description of Dynamic Programming Algorithms

Tesis que presenta el **M. en C. Arturo Díaz Pérez** para obtener el grado de **Doctor en Ciencias** dentro de la especialidad en **Ingeniería Eléctrica** con opción en **Computación**. Trabajo dirigido por el Dr. Guillermo Morales Luna.

México, D. F. 31 de julio de 1998

A Mary,

Carmelita,

y Lilí

A mis padres,

a mis hermanos,

y a mis sobrinos

ACKNOWLEDGEMENT

Many people have been involved in the elaboration of this thesis. To all of them the only thing I can say is thank you. However, some people deserve special considerations from me:

to Guillermo Morales Luna for having confidence on my research and helping me during all the stages of this investigation. For accepting to mentor me and teach me, for working with me on a day to day basis, and for being a very good friend;

to Mike Quinn for accepting me at his lab, for being willing to work with me under unusual circumstances during that unforgettable year I spent at OSU. His multiple comments and opinions have greatly influenced the results of this research;

to Sanjay Rajopadhye for introducing me the fundamental theory of systolic arrays;

to Sergio Rajsbaum, Adriano de Luca, Bill Taffe and Isidoro Gitler for accepting to serve on my committee and for the time they spent in my behalf;

to Sergio Chapa for all support he offered me during my stay at the Sección de Computación;

to CINVESTAV for giving me facilites and support to finish this investigation;

to my little girls Carmelita and Lilí for the time and love we sacrificed because of this research;

and finally to my wife Mary for her constant love and unconditional support.

This work was undirectly supported by CONACyT with the scholarship 68102.

Resumen

Los arreglos de compuertas programables (FPGAs por sus siglas en inglés), se han usado de manera exitosa como procesadores configurables en algunas aplicaciones. Algunas aplicaciones han rebasado el rendimiento ofrecido por sistemas de cómputo de propósito general. Desafortunadamente, la tecnología actual de FPGAs presenta una serie de limitaciones de manera que éstos no pueden aplicarse de forma arbitraria para cualquier tipo de algoritmos. Para implementar algoritmos en FPGAs uno debe ir desde una descripción de alto nivel del algoritmo hasta secuencias de digitos binarios (bits) que representa el diseño para configurar uno o posiblemente varios FPGAs. La mayoría del trabajo actual se ha enfocado a los aspectos de bajo nivel de este enfoque, esto es, al desarrollo de herramientas para generar diseños de circuitos a implementarse en FPGAs. La entrada de este enfoque es usualmente, una descripción de un algoritmo a un nivel muy bajo. En esta tesis, presento una metodología de diseño para implementar algoritmos en FPGAs a partir de descripciones de alto nivel. Dado que los FPGAs no pueden ser aplicados para implementar algoritmos arbitrarios, esta metodología se restringe a la clase de algoritmos de programación dinámica cuya importancia ha sido probada de manera extensa. La metodología parte de un algoritmo de programación dinámica escrito en el lenguaje de programación C* y termina produciendo un diseño de un circuito en el lenguaje de descripción de circuitos VHDL. En las etapas intermedias, se obtienen las ecuaciones de recurrencia que describen el comportamiento funcional del programa y esta ecuaciones se llevan, cuando es posible, a una forma uniforme. A partir de ecuaciones uniformes se obtienen representaciones que configuran un arreglo de procesadores (espacio) en diferentes instantes de tiempo. El diseño de un circuito en VHDL se obtiene a partir de las representaciones espacio-tiempo.

Abstract

Field programmable gate arrays (FPGAs) have been successfully applied as computing engines for some applications. Some of them have outperformed large-scale general-purpose computing systems. However, current FPGA technology imposes a number of limitations that severely restrict the class of algorithms FPGAs are useful for. In implementing algorithms for FPGAs one must go from an algorithmic description to binary files which represent a design to configure one, or possibly more, FPGAs. Most of the current work has been focused on the low-level aspects of this approach, i.e., in the tools to generate circuit design to implement algorithms in FPGAs. The input of this approach is usually an enough low-level description of an algorithm. In this thesis, I present a design methodology whose main concern is on the high-level aspects of algorithm implementation for FPGAs. Since currently FPGAs cannot be used for implementing arbitrary algorithms, I restrict myself to the class of dynamic programming algorithms. The importance of dynamic programming has been proved elsewhere. I start from a dynamic programming program written in a parallel language, C*, and a circuit design in a hardware description language, VHDL, is produced. In the middle, it is obtained the recurrence equations that describe the functional behavior of the program, these equations are transformed to a uniform shape, and, finally, space-time descriptions are produced from which the design in VHDL is generated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Performance is a continuing concern in the design of computer-based systems. Advances to enhance performance can be broadly placed into two main categories: technological and architectural. Technological advances involve finding new materials and techniques to make gates that switch faster and memories that can be accessed faster. Architectural advances involve reorganizing these gates and memories to allow more operations to occur at the same time. Until the mid 90's, technological advances dominated increases in speed. However, the technology is approaching fundamental limits. Future increases in performance will be forced to rely more heavily on advances in computer architecture.

One of the primary methods for gaining performance-improvement based on the possibilities of the latest VLSI technologies is to migrate functions or parts of functions to auxiliary special purpose processors and controllers dedicated to a specific application area or algorithm. The first examples of such an approach were application-specific integrated circuits (ASICs). Their applications are varied. They rank from digital signal processing applications to neural networks implementations and special controllers. However, ASICs are inflexible which has limited their use. Moreover, the time and cost to design and build an ASIC have prevented its applicability.

Some kind of circuits were proposed for general-purpose applications. The most successful examples of that are probably the VLSI array processors used for signal processing [50]. Although array processors have several advantages, ASICs restrictions have limited their use. Those restrictions have been partially broken with the rise of field-programmable gate arrays (FPGA) [63]. Currently, FPGA technology provides an alternative for implementing application-specific circuits without the non-recurring engineering cost associated with ASICs.

An FPGA is an IC consisting of user-programmable logic blocks (CLBs) and interconnection fabric that can be used in the design of a digital circuit [73]. Any kind of digital circuit can be implemented using FPGAs. Each block in the circuit is user-programmable, meaning that the functionality implemented by the block can be determined by the user after the block has been fabricated. Many FPGAs providers exist, among the best known are Xilinx [82], Altera [5], Actel [1] and Algotronix [4].

FPGAs were first applied to logic design. Hence numerous tools were provided for synthesizing logic designs for FPGAs. Hardware description languages (HDL) and diagram editors have been used as a starting point for such a process. Many hardware description languages exist; each company has developed its own. Verilog and VHDL are among the most popular languages we can find [59]. To avoid implementing every design from scratch, most providers offer several libraries of predefined devices or modules from which a design can be built.

Recently, FPGAs have been used for custom computing where the goal is obtaining improvements in performance by implementing algorithms in hardware at reasonable cost. The main advantage of FPGAs is their reconfigurability, i. e., they can be used for different purposes at different stages of a computation. Numerous reconfigurable architectures based on FPGAs have already been built [8, 14, 28, 36]. Some of them have outperformed large-scale general-purpose computing systems for some applications [36]. Those applications have demonstrated that FPGAs can be integrated in scalable architectures.

Each individual CLB of an FPGA has a limited logical complexity. Although FPGA density has greatly increased in recent years[*], the aggregate non-integral compute power of an FPGA chip is relatively low. Current FPGA technology imposes several limitations that severely restrict the class of algorithms FPGAs are useful for. DeHon [25] provided examples of algorithms well suited for reconfigurable engines. For this reason, only some sets of applications—those working at the bit level or using fixed point arithmetic—can lead to levels of performance superior to conventional, readily available CPUs. An approach to overcome with this limitation is to have reconfigurable computing engines organized with arrays of FPGAs. That exhibits new restrictions to make effective use of FPGA area. Local and regular communications are desirable properties of FPGA algorithmic applications.

In spite of the current limitations, I believe that FPGA density will increase in the future to allow more complicated functions be implemented on them. When that technology eventually becomes available, tools and techniques will be required to effectively use FPGAs as custom computing engines.

As the number and the density of FPGAs increase, the task of developing custom circuits for each FPGA in the system becomes enormous. In addition, the lack of knowledge and tools necessary to develop reconfigurable applications further obstructs general purpose implementation. A strong background in hardware development is required as well as expensive CAD and synthesis tools. Until reconfigurable systems address the deficiencies of large-scale application development, reconfigurable logic will remain in the application-specific realm.

In order for reconfigurable systems to become general purpose computing systems, they must be easy to program and use. Although some early work has been done on automated software/hardware co-synthesis [2], most reconfigurable systems are programmed using conventional hardware development techniques such as schematic capture or hardware description languages. However, hardware description languages are very low level languages for describing algorithms, therefore, most of the custom computing research has been focused on the low-level aspects of algorithm implementation in FPGAs.

One way to reduce the problem of realizing custom circuitry on reconfigurable hardware systems is to develop tools and techniques for the hardware implementation of algorithms starting from high level descriptions. That involves on the one hand, developing compiling techniques for the synthesis of programs to hardware description languages, and on the other hand, characterizing the applications susceptible to take advantage of that approach. This research considers the high-level aspects of algorithm implementation for reconfigurable engines and address these two questions providing a general framework to face those problems.

The main objective of this dissertation is to present a methodology for the synthesis of algorithms from high-level specifications to hardware description level. It is considered, on the one hand, the limitations of FPGAs, and on the other hand, some techniques developed for synthesizing processor arrays.

I propose to use the C* parallel programming language as a high level specification of programs to be implemented in FPGAs and to use VHDL as a target language as it is shown in Figure 1.1. C* is based on a synchronous SIMD model (data-parallel) of computation with a global name addressing mechanism. VHDL has become a standard language for circuit implementation. On one side, previous research has demonstrated that data-parallel programs can be compiled and efficiently executed on both SIMD and MIMD parallel computers [40]. On the other side, numerous commercial tools exist to implement circuits from VHDL.

My main interest here is to determine how to apply the data-parallel paradigm to FPGA computing. Some other works, dealing with bit-level applications, have tried to develop compilation techniques to derive hardware descriptions from high level programming languages [37, 8]. Considering the FPGA scalability and characteristics, my goal here is showing how to derive regular processor arrays for FPGA computing from some data-parallel algorithms expressed in the C* programming language.

[*]Altera has announced a new series of FPGA having until 200K reconfigurable gates

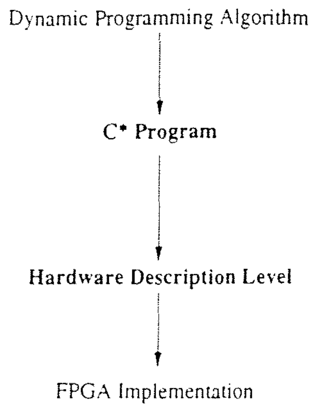


Figure 1.1: Translating C* to hardware description level.

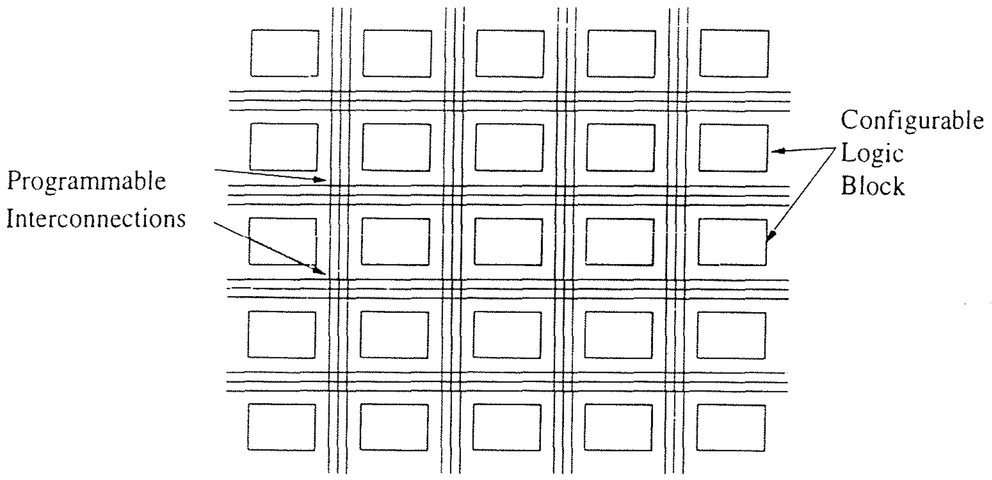


Figure 1.2: A typical view of an FPGA.

1.1 FPGA Characteristics and Limitations

As was pointed out before, the main feature of FPGAs is their re-programmability. FPGAs are general-purpose circuits capable of implementing any digital circuit. The logic blocks of FPGAs may consist of combinatorial or sequential logic elements. The blocks can be connected after fabrication in a general way using a programmable routing fabric. This routing fabric consists of programmable switches that connect various horizontal and vertical routing segments together with the pins from logic blocks [78] (See Figure 1.2). The logic blocks architecture available in commercial FPGAs ranges from fine grain to coarse grain blocks. The interconnection re-programmability is based on either static RAM or antifuse technology [38].

In Figure 1.3 we show the CLB of the Xilinx 4000 FPGA series. Each CLB contains a pair of flip-flops and two independent 4-input function generators. The flip-flops can be used to store the function generators outputs or they can be used independently. A third function generator with three inputs is provided. One or both of these inputs can be the outputs of the 4-input function generators; the other input(s) can be from outside the CLB. The CLB can therefore implement any of the following functions: (1) two functions of up four variables, plus any third function of up three unrelated variables, (2) any single function of up five variables, (3) any function of four variables together with some functions of six variables, and (4) some functions of up nine variables.

Implementing wide functions in a single block reduces both the number of blocks required and the delay in the signal path, achieving both increased density and speed. The versatility of the CLB

C1
C2
C3
C4
H1
DIN
S/R
EC
S/R CONTROL
G4
G3
G2
G1
LOGIC FUNCTION OF G1-G4
G
DIN
F
G
H
D
SD
Q
YQ
LOGIC FUNCTION OF F', G', AND H1
G
H
EC
RD
Y
S/R CONTROL
F4
F3
F2
F1
LOGIC FUNCTION OF F1-F4
F
DIN
F
G
H
D
SD
Q
XQ
EC
RD
K (CLOCK)
H
F
X

Figure 1.3: The configuration logic block of the Xilinx 4000 FPGA Series.

function generators significantly improves system speed. In addition, design-software tools can deal with each function generator independently which improves cell usage.

The XC4013E model of Xilinx FPGA series contains a matrix of 24 × 24 CLBs. That produces 567 CLBs which can store up to 13,000 gates and 1,536 flip-flops. In table 1.1 we show some common circuit functions, the number of CLBs they occupy and the speed of the circuit.

As can be observed, the density and speed of different functions is varied. Currently, the main disadvantage is the low density of logic blocks per area unit available with present VLSI technology. Due to this there are some interesting circuits that can not be successfully implemented for array processors. For example

- Floating-point arithmetic is severely limited for FPGAs because it would require wide area on the circuit. Trying to implement arrays of floating point functional units would not be easy.
- Large and variable interconnectivity would be restricted due to delays involved in the interconnection architecture of FPGAs.
- Extremely fast applications possibly would not be suitable for FPGAs due to delays present in their architecture.

Regardless above restrictions, FPGAs can be properly used for rapid prototyping algorithms at the hardware level. Considering the restrictions of FPGA applications, the desirable properties of potential FPGAs applications would be:

1. To use only integer arithmetic or at most low precision fixed point arithmetic.
2. To consist of logical operations to make decisions. Comparators, selectors and multiplexers are good examples of that.
3. To be capable of being decomposed in independent and pipelined stages.
4. To have regularity in the way processing is applied.

Design Class	Function	CLB used	Speed/Units
Memory	256x8 Single Port (read/modify/write)	72	63 Mhz
Memory	32x16 bit FIFO (simultaneous read/write)	48	63 Mhz
Memory	32x15 bit FIFO (MUXed read/write)	32	63 Mhz
Logic	9 bit Shift Register (with enable)	5	170 Mhz
Logic	16 bit Pre-Scaled Counter	8	142 Mhz
Logic	16 bit Loadable Up/Down Counter	8	70 Mhz
Logic	16 bit Accumulator	9	70 Mhz
Logic	8 bit, 16 tap FIR filter (parallel)	400	55 Mhz
Logic	8 bit, 16 tap FIR filter (serial)	68	8.1 Mhz
Logic	8x8 Combinatorial Parallel Multiplier	68	22.8 ns
Logic	16 bit Address Decoder (internal decode)	3	4.7 ns
Logic	9 bit Parity Checker	1	4.3 ns

Table 1.1: Density and performance for common circuit functions

5. To have locality in the interconnection network they require. That means that they have only interconnections with their neighbors.

Considering FPGA capabilities and limitations some potential applications for FPGAs are:

1. Image processing algorithms such as point type operations (grey scale transformation, histogram equalization, requantization, etc.) and filtering (template matching, window techniques, convolution/correlation, median filtering, etc.) seem to be good candidates for FPGA implementation.

2. Dynamic programming algorithms requiring only integer arithmetic. Dynamic programming is in essence a bottom up procedure in which solutions to all subproblems are first calculated and the results used to solve the whole problem. A good example of this approach is the Floyd's shortest path algorithm.

3. Relaxation techniques requiring fixed point arithmetic. The relaxation technique is an iterative approach to many problems, which makes updating in parallel at each point and in each iteration based on the data available in the most recent updating or in the immediate preceding iteration.

4. Associative retrieval operations. Filling and retrieving data by association appears to be a powerful solution to many high volume information processing elements. An associative processing system is very adequate for recognition and recall from partial information and has remarkable error correcting capabilities. The major advantage of associative memory over RAM is its capability of performing parallel search and parallel comparison operations. There many examples of that kind of applications: pattern matching, artificial intelligence, computer vision, data encoding, compression, and every application maintaining a dictionary data structure.

1.2 Custom Computing Review

The general idea of custom computing is to add a co-processor to a common processor to serve in the execution of some parts of a program. Given that co-processors built with FPGAs are reconfigurable, then such co-processors can be configured to execute different functions at different times of the program execution. Several custom computing machines have been proposed. They can be classified in one of three different approaches as it is shown in Figure 1.4. In Figure 1.4a we show the most common configuration for a custom computing machine. An array of FPGAs is

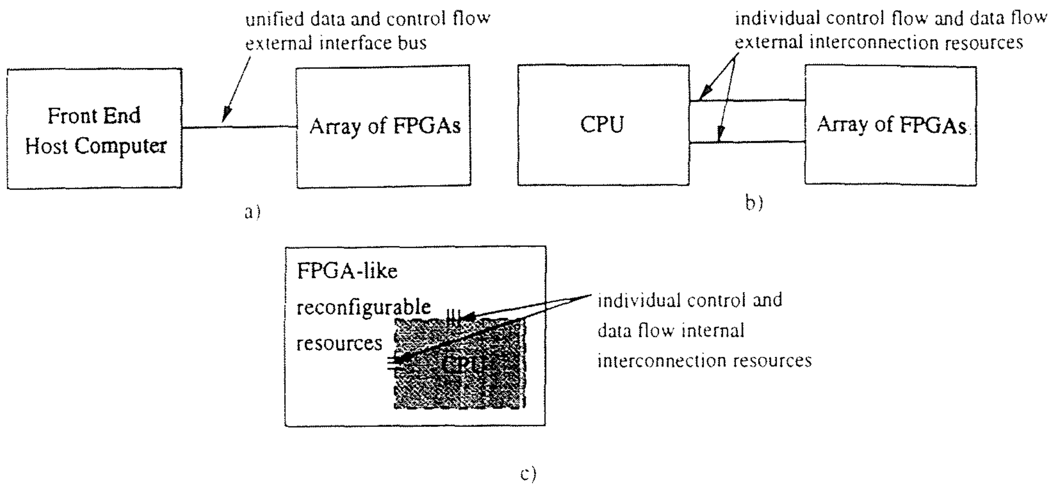

unified data and control flow external interface bus
Front End Host Computer
Array of FPGAs
a)
individual control flow and data flow external interconnection resources
CPU
Array of FPGAs
b)
FPGA-like reconfigurable resources
CPU
individual control and data flow internal interconnection resources
c)

Figure 1.4: Approaches to custom computing.

used within a host computer; they are configured through the internal bus. In the same manner, control signals, data and results flow through the host bus. Performance is limited here by the bus bandwidth to transfer data. The best known and most successful machines have this configuration. Some examples are Splash [36], DEC Perle [28], and AnyBoard [14].

In a natural way, it has been proposed to separate control and data streams using different buses. That is shown in Figure 1.4b. Some examples of this configuration are PRISM developed at Boston University [8], and the University of Toronto's reconfigurable processor. Performance is still limited by the data and control buses bandwidth.

A third approach, showed in Figure 1.4c, is to have a circuit with a sea of reconfigurable gates in which a part is dedicated to perform the function of a conventional processor and the rest is devoted to implement algorithms through reconfigurable logic. It is intended to eliminate bandwidth restrictions. However, it is still at research level and no prototype is known to be already implemented [25].

1.3 Our Approach

In implementing algorithms for FPGAs one must go from an algorithmic description to binary files which represent a design to configure one, or possibly more, FPGAs. Hardware description languages, although useful for digital design, are very low level languages for describing algorithms. Moreover, although much effort has been devoted to develop tools for the automatic synthesis of designs expressed in hardware description languages to FPGAs, there is a lack of techniques for synthesizing high level algorithmic descriptions to low level descriptions for FPGA computing.

In this research we present a design methodology to synthesize algorithms for FPGAs. We consider, on the one hand, the limitations of FPGAs, and on the other hand, the techniques developed to synthesize regular processor arrays [68]. We start from a dynamic programming algorithm expressed in a C* program and we translate it to hardware description level in VHDL. Once we get hardware level descriptions we can use standard tools to synthesize a design for FPGAs.

The process of synthesis to hardware description level is illustrated in Fig. 1.5. From nested loops containing data-parallel statements in C* we obtain systems of recurrence equations (SRE) which are translated, when possible, into systems of uniform recurrence equations (SURE). Although there is no a general strategy to transform a SRE into a uniform shape, we explore how to address that problem. Decisions about variable alignment, operation serialization and interprocessor communication localization will be done through an interactive tool that applies transformations to the equations obtained.

Once we have generated a SURE we can explore different scheduling and allocation functions.

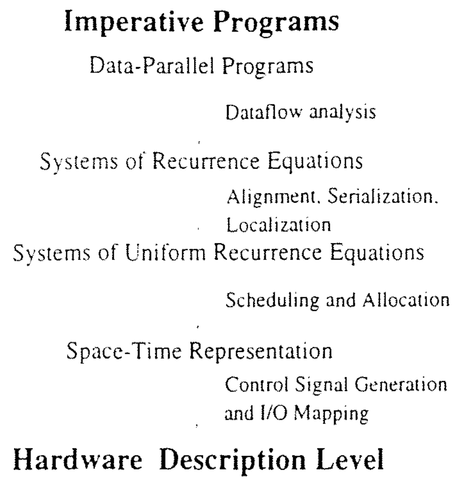

Imperative Programs
Data-Parallel Programs
Dataflow analysis
Systems of Recurrence Equations
Alignment, Serialization, Localization
Systems of Uniform Recurrence Equations
Scheduling and Allocation
Space-Time Representation
Control Signal Generation and I/O Mapping
Hardware Description Level

Figure 1.5: Translation C* to hardware description level.

Linear transformations can be applied to SUREs to obtain a space-time representation from which we can derive the structure of a systolic array. The hardware level description of the array is found after generating the necessary control signals for the proper behavior of the array and after mapping the I/O channels onto the array.

1.4 Summary of Major Contributions

Following is a summary of the important research contributions:

1. Determination of a class of algorithms for which FPGAs can be adequately used to obtain better performance than that which can be obtained in general-purpose computers. Current FPGA density prohibits their use for general purpose algorithms. The restrictions of FPGA technology are considered in determining the type of algorithms which can be benefited of being implemented in hardware.

2. A general design strategy to implement that class of algorithms through the data-parallel paradigm is proposed. As a first step, the C* programming language is used as a high level description for the class of algorithms determined in step 1. Design strategies are proposed for implementing that class of algorithms for two target architectures: FPGA-based computing engine, and a general-purpose parallel computer.

3. Compilation tools to transform data-parallel programs in suitable form for hardware synthesis. A combination of automatic and manual techniques are provided for developing hardware level descriptions of data-parallel programs.

4. A generic behavioral hardware description model is proposed which is based on reconfigurable hardware and is adequate to implement the class of algorithms determined in step 1.

5. All the previous parts are integrated to provide a general framework for the implementation of algorithms in reconfigurable hardware based on FPGAs. I will propose a general framework to the synthesis of algorithms for FPGA computing.

1.5 Overview of the Thesis

The thesis is organized in 8 chapters. In chapter 2, we discuss dynamic programming problems and we present a classification of them. In chapter 3, we present a new approach to implement a class of DP problems in a general-purpose multicomputer. In chapter 4 we introduce the concept of recurrence equations which are fundamental to understand the rest of the work. In chapter 5, we develop an algorithm to extract recurrence equations from C* program fragments. Chapter 6 is devoted to discussing the uniformization process of recurrence equations and a tool to perform such a job is described. In chapter 7 we present an approach to generate hardware level descriptions from space-time representations. Finally, conclusions are drawn in Chapter 8.

Chapter 2

A Classification of Dynamic Programming Algorithms

2.1 Introduction

Dynamic programming (DP) is a general problem solving strategy often applied to optimization problems. The theory of dynamic programming was originally introduced by Richard Bellman to solve mathematical problems arising from multistage decision processes [12]. Dynamic programming has found numerous applications in computer science such as: optimal parenthesization [39], line and curve detection [22], parsing of general context-free languages [21, 17], string matching [54, 55], hand-written symbol recognition [20]. Its application was particularly successful in speech recognition [69, 75, 62, 79], and in image processing and computer vision [61, 9]. These are also applications in which the running time is critical and in which parallel processing is required.

Many different approaches have been proposed to implement dynamic programming algorithms. Some of them have tried to obtain general strategies for deriving implementations of various problems. However, currently there is not a general strategy to implement dynamic programming. To understand the nature of DP problems we need to make some observations about their formulation.

In this chapter we explore general formulations of DP problems based on three type of classifications. In the first type we determine if the formulation involves just one subproblem, a fixed number of subproblems or a non-fixed number of subproblems. In the second aspect we will consider whether the recursive formulation is inherently sequential or can be decomposed to ignore sequentiality. In the third point of view, we will consider the dimensionality of the DP table as a clue to the complexity order of the problem.

In section 2.1 we will review the formulation of dynamic programming problems. We will consider the Principle of Optimality that all DP problems must obey. In section 2.2 we review the classification of DP problems and we will extend that classification considering important aspects of the formulations for its implementation. In section 2.3 we will present a set of problems that illustrate each of the classes we will consider. A discussion about implementation issues can be found in chapter 3.

2.2 Dynamic Programming Formulations

Dynamic programming (DP) is a general problem-solving strategy often applied to optimization problems. The general formulation can be stated as follows: let

$$P^o : p^* = opt_{x \in X} q(x) \tag{2.1}$$

be an optimization problem where X is the *solution set* and q is a real valued function defined as the *objective function.* Sneidovich [77] showed that P^o will yield a dynamic programming optimality equation if the solution set X is a subset of the Cartesian product of two sets, $Y \times Z$, leading to reformulate the problem in the following manner:

$$p^* = opt_{(y,z)\in X}q(y,z), \qquad X \subset Y \times Z \tag{2.2}$$

Now, if we express X as the union of bundles $\{y\} \times Z(y)$ where $y \in Y$ and $Z(y) \subset Z$, i. e. $X = \bigcup[y \times Z(y)]$, we can reformulate Eq. (2.2) as follows:

$$p^* = opt_{y\in Y}p(y) \tag{2.3}$$

where,

$$\forall y \in Y : p(y) = opt_{z\in Z(y)}q(y,z) \tag{2.4}$$

The decomposition of the original problem into two optimization problems is due to the *Principle of Optimality* [13], which states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision. An effort devoted to the rigorous mathematical framework and effective evaluation of DP problems can be found in [77].

Clearly, dynamic programming problems can be formulated using a recurrence relation involving a decision process. Dynamic programming decomposes a problem into a number of smaller subproblems, each of which is further decomposed until subproblems have trivial solutions. For example, a problem of size n may decompose into several problems of size $n - 1$, each of which decomposes into several problems of size $n - 2$, etc. This decomposition seems to lead to an exponential-time algorithm, which is indeed true for some problems, such as the traveling salesman problem. However, in many other problems there are only a polynomial number of distinct subproblems. Dynamic programming gains its efficiency by avoiding solving common subproblems many times. It keeps track of the solutions of subproblems in a table, and it performs table lookup whenever needed.

From the algorithmic design point of view, the development of a dynamic programming strategy has four steps: (1) characterize the structure of an optimal solution, (2) recursively define the value of an optimal solution, (3) compute the value of an optimal solution in a bottom-up fashion, and (4) construct an optimal solution from the computed information [24]. Once this is known, a two-step dynamic programming algorithm emerges: (1) construction of a table in a bottom-up fashion which contains solutions to smaller subproblems of the given problem (*forward step*), and (2) construction of a solution for the given problem (*backward step*). The big challenge in dynamic programming strategies is how to build the DP table.

Each entry of a DP table corresponds to a subproblem. Thus, the size of the table is the total number of subproblems including the problem itself. Because of the recurrent formulation, each entry of the table depends on one or several subproblems. Therefore, the order to fill the table may be chosen under the restriction of the table and the entry dependency. For sequential implementations there is an obvious algorithm which fills the table according to the entry dependency. To exploit parallel processing, we must examine the recurrence formulation.

2.3 Classification of DP Algorithms

In this section we will unify the classifications done by Li and Wah [53] and Galil and Park [35].

Li and Wah classified DP problems according to the form of their functional equations and the nature of the recursion. That classification was first used to derive some systolic algorithms. Later on, Kumar [49] used the same classification to show practical implementations of problems belonging to each class.

In general, the solution to a DP problem is expressed as a minimum (or maximum) of possible alternative solutions. Each of these alternative solutions is constructed by composing one or more

	serial	**non-serial**
monadic	● shortest-path ● 0-1 knsapsack	● longest common subsequence ● GAP problem ● RNA problem ● Least weight subsequence
polyadic	● all-pairs shortest paths	● optimal matrix parenthesization

Figure 2.1: Dynamic programming problems classification and examples.

subproblems. A DP formulation is called *monadic* if its composition function involves only one recursive term; otherwise it is called *polyadic*. A DP formulation is *serial* if the subproblems can be grouped in levels, and the solution to any subproblem in a certain level can be found using subproblems that belong only in the immediately preceding levels. Otherwise it is *non-serial*. As shown in [53, 49] *monadic-serial* DP problems can be solved by a series of matrix-vector multiplications which is easy to parallelize. On the other hand, there is no general parallel formulation for *polyadic-serial* DP problems. In Figure 2.1 some example problems are presented according to their formulation.

Galil and Park [35] classified DP problems according the size of the table and the number of dependencies in a single problem. A dynamic programming problem is called a tD/eD problem if its table size is $O(n^t)$ and a table entry depends on $O(n^e)$ other entries. Although Galil and Park did not discuss the importance of the classification, the number of dependencies is of relevance to derive efficient implementations.

In Table 2.1 we present a number of DP algorithms and their characteristics according to the preceding discussion. We can see that problems with more than $O(1)$ dependencies can be monadic or polyadic, serial or nonserial. We can make various observations of the characteristics of DP problems in considering them to derive efficient implementations.

As it will be evident in the next sections, the fact that a problem is serial or nonserial affects the way in which we build the DP table. In a serial problem, only the current and the previous level subproblems need to be stored. In a nonserial problem, solved subproblems on all previous levels are required. When we write an imperative program to fill the table of a nonserial problem, we need a scheduling policy.

Whether a problem is monadic or polyadic influences the way in which the solution of a specific problem is built (the second stage in the algorithmic point of view). Solutions of monadic problems are built in a serial fashion. Solutions of polyadic problems can be built by a divide-and-conquer strategy.

2.4 Some Examples of DP Problems

In this section we will present some problems and we will discuss their formulations and classifications.

2.4.1 The Shortest-Path Problem

The solution to the shortest-path problem was the original dynamic programming strategy proposed by Bellman [12]. It considers computing the minimum path between a source and a destination point

Problem Formulation			**Deps.**	**T. Size**
Shortest Path[24] $C_i^l = \min_{1\le j\le n}\{c_{ij}^l + C_j^{l+1}\}$	*monadic*	*serial*	$O(n)$	$O(n^2)$
0-1 Knapsack[49] $F[i,x] = \begin{cases} x \ge 0, i=0 \rightarrow & 0 \\ x < 0, i = 0 \rightarrow & -\infty \\ 1 \le i \le n \rightarrow & \max\{F[i-1,x], \\ & F[i-1, x-w_i] + p_i\} \end{cases}$	*monadic*	*serial*	$O(1)$	$O(n^2)$
Longest Common Subsequence[24] $F[i,j] = \begin{cases} i=0 \vee j=0 \rightarrow & 0 \\ i,j>0 \wedge a_i = b_j \rightarrow & F[i-1,j-1]+1 \\ i,j>0 \wedge a_i \ne b_j \rightarrow & \max\{F[i-1,j], \\ & F[i,j-1]\} \end{cases}$	*monadic*	*nonserial*	$O(1)$	$O(n^2)$
Edit Distance[24] $F[i,j] = \begin{cases} i=0 \vee j=0 \rightarrow & 0 \\ i=0 \wedge j>0 \rightarrow & D[0,j-1]+K_o \\ i>0 \wedge j=0 \rightarrow & D[i-1,0]+K_a \\ i,j>0 \rightarrow & \min\{D[i-1,j-1]+d(t_i,r_j), \\ & D[i,j-1]+K_o \\ & D[i-1,j]+K_a\} \end{cases}$	*monadic*	*nonserial*	$O(1)$	$O(n^2)$
Gap[33] $D[i,j] = \min \begin{cases} D[i-1,j-1]+s_{ij} \\ \min_{0\le q<j}\{D[i,q]+w(q,j)\} \\ \min_{0\le p<i}\{D[p,j]+w'(p,i)\} \end{cases}$	*monadic*	*nonserial*	$O(n)$	$O(n^2)$
RNA[76] $D[i,j] = \min_{0\le p<i \;\; 0\le q<j}\{D[p,q]+w(p,q,i,j)\}$	*monadic*	*nonserial*	$O(n^2)$	$O(n^2)$
Least Weight Subsequence[42] $D[j] = \min_{0\le i<j}\{D[i]+w(i,j)\}$ for $1 \le j \le n$	*monadic*	*nonserial*	$O(n)$	$O(n)$
All Pairs Shortest Paths[24] $d_{ij}^{(k)} = \begin{cases} k=0 \rightarrow & w_{ij} \\ k\ge 1 \rightarrow & \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \end{cases}$	*polyadic*	*serial*	$O(1)$	$O(n^3)$
Optimal Matrix Parenthesization[24] $m(i,j) = \begin{cases} 1\le i<j\le n \rightarrow & \min_{i\le k<j}\{m(i,k)+ \\ & m(k+1,j)+ \\ & r_{i-1}r_kr_j\} \\ j=i, 0\le i\le n \rightarrow & 0 \end{cases}$	*polyadic*	*nonserial*	$O(n)$	$O(n^2)$

Table 2.1: A set of dynamic programming algorithms.

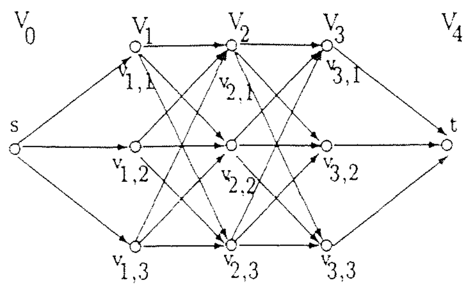


Figure 2.2: A multistage graph for the shortest-path problem.

where the path passes a number of stages, in each stage several possibilities exist as it is shown in Figure 2.2. Let $G = (V, E)$ be a multistage graph in which V is the set of vertices and E is the set of edges. $V = V_0 \cup V_1 \cup \cdots \cup V_N$, where V_i, $0 <= i <= N$ is the set of nodes in stage i, $V_0 = \{s\}$ and $V_N = \{t\}$, and v_{ij} is the j'th node in V_i. We assume that there exist exactly m nodes in each stage other than 0 and N. The cost of a path from the source node, s, to sink node, t, is the sum of costs on the edges of the path. Let P be the set of all possible paths from s to t. A path $p \in P$ is of the form $s - v_{1j_1} - v_{2j_2} - ... - v_{n-1j_{n-1}} - t$. If the cost of edge $(v_{ij_i}, v_{i+1,j_{i+1}})$ is $g_i(v_{ij_i}, v_{i+1j_{i+1}})$ the minimum-cost path from s to t is:

$$\min_{p \in P} f(p) = \min_{p \in P} \sum_{i=0}^{N-1} g_i(v_{ij_i}, v_{i+1j_{i+1}}) \tag{2.5}$$

Each term in optimization problems of this form shares one variable with its predecessor term and another one with its successor term. This problem has a serial structure, and, consequently, is called a serial optimization problem. Many practical problems can be formulated in this way: traffic-control, circuit-design, fluid-flow and scheduling.

We can reformulate the problem as follows. Let us assume for simplicity that every stage consists of exactly m nodes. Let $c^k(j), 0 \leq k \leq N, 1 \leq j < m$, be the minimum cost of going from node j in stage k to the sink node, t. We can group the $c^k(j)$ values into a vector of the following form

$$C^k = [c^k(0), c^k(1), ..., c^k(m-1)]^T \tag{2.6}$$

Let $M^k = g_k(i, j)$ be the cost of going from node i, in stage k, to node j in stage $k+1$, $1 \leq k < N$. M^k is a $m \times m$ matrix. It can be seen that

$$C^k = M^k * C^{k+1}, 1 \leq k < N \tag{2.7}$$

which is a matrix-vector multiplication where products can be interpreted as additions and additions as minimum selection operations. This formulation induces a sequence of $N - 1$ matrix-vector-style multiplications. Then, the solution of the problem can be easily computed from the dot product

$$C^0 \cdot C^1 = [g_0(0), g_0(1), ..., g_0(m-1)]^T \cdot C^1 \tag{2.8}$$

where, $g_0(j)$ is the cost of going from source node, s, to node j in stage 1. Equations 2.7 and 2.8 provide us a recurrent formulation for this problem. Since the composition function involves only one recursive term, this problem is monadic. For obvious reasons it is serial. It presents a fixed number of dependencies and its table is of size $N \times m$.

2.4.2 The Knapsack Problem

This is a classical problem used to illustrate DP algorithmic formulations. It has received great attention for many researchers looking for efficient implementations. See [7] for a novel approach.

Suppose that m types of objects are being considered for inclusion in a knapsack of capacity c. For $i = 1, 2, \cdots, m$, let p_i be the *profit* and w_i the *weight* of the i-th type of object, where w_i, p_i, and c are all positive integer. The knapsack problem is to choose a collection of objects in such a way that the total profit without exceeding the capacity is maximized, i.e.,

$$\max \left\{ \sum_{i=1}^{m} p_i z_i : \sum_{i=1}^{m} w_i z_i \leq c, z_i \in \mathbb{N}, i = 1, 2, \cdots, m \right\} \tag{2.9}$$

where z_i is the number of i-th type objects included in the knapsack. The problem, as specified above is often called the *unbounded knapsack problem*, since the only constraint on the solution (other than the capacity constraint) is that it is non-negative. There are many variations of the problem, such as the *bounded knapsack problem* (here, additional constraints of the form $z_i \leq b_i$ must be satisfied), the 0/1 *knapsack problem* (a particular case of the bounded problem, where $b_i = 1$: there is exactly one copy of each type of object), the *subset sum problem* (a 0/1 problem with $w_i = p_i$), the *change making problem*, etc. They arise in different application domains, and are all NP-complete.

Let us define the function $f(j,k)$ which denotes the value of an optimal solution of the sub-problem where only the first k objects are considered, and only a capacity of j is available. It is well known that the computation of $f(i,j)$ is specified recursively as follows:

$$f(i,j) = \begin{cases} 0 & \text{if } j = 0 \text{ or } k = 0 \\ f(j, k-1) & \text{if } k > 0 \text{ and } j < w_k \\ f(j, k-1) \oplus (p_k + f(j - w_k, k - \beta)) & \text{if } k > 0 \text{ and } j \geq w_k \end{cases} \tag{2.10}$$

where $f(c,m)$ is the solution to the problem. The different variations of the knapsack problem merely correspond to different choices of the operator $\oplus$ and the constant β in Eq. 2.10: in the unbounded knapsack problem, $\oplus$ is max, and $\beta = 0$; in the 0/1 and subset-sum problems, $\oplus$ is max and $\beta = 1$; in the change making problem $\oplus$ is min and $\beta = 0$.

An elegant, memory efficient implementation for the problem can be found in [43]. It is computed in time $O(c+m)$ time and c space (only the last column needs to be saved, not the entire table). Because of the dynamic dependencies existing in the knapsack problem, this problem has been heavily studied. One instance of this problem can be observed in Figure 2.3. This is another monadic-serial algorithm whose table is m^2 having constant non-fixed dependencies.

2.4.3 Longest Common Subsequence

The longest-common subsequence problem belongs to a class of widely studied problems related to string-matching [76]. Finding the longest-common subsequence of two strings can be stated as follows: a subsequence $z = \langle z_1, z_2, \cdots, z_k \rangle$ is a subsequence of $x = \langle x_1, x_2, \cdots, x_n \rangle$ if there exists a strictly increasing sequence $\langle i_1, i_2, \cdots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \cdots, k$, we have $x_{i_j} = z_j$. A common subsequence of A and B is a sequence Z that is a subsequence of both A and B. For example the sequence $\langle B, C, B \rangle$ is a common subsequence of $\langle B, D, C, A, B, A \rangle$ and $\langle A, B, C, B, D, A, B \rangle$. Given two sequences $A = \langle a_1, a_2, \cdots, a_n \rangle$ and $B = \langle b_1, b_2, \cdots, b_m \rangle$ we wish to find a maximum-length common subsequence of A and B. For the previous example the longest common subsequence is $\langle B, C, B, A \rangle$.

The dynamic programming formulation of the problem computes $F[i,j]$ $(0 \leq i \leq n, 0 \leq j < m)$, where $F[i,j]$ denotes the longest common subsequence of the first i elements of A and the first j elements of B, and it is defined by the following recurrence equation.

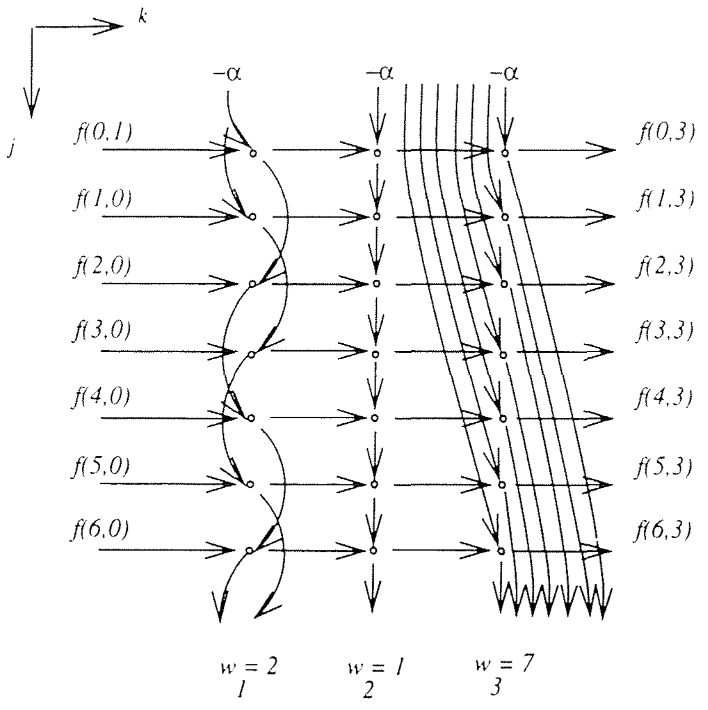


Figure 2.3: Dependencies of the knapsack problem.

		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

Figure 2.4: An example of the longest common subsequence problem.

$$F[i,j] = \begin{cases} i = 0 \vee j = 0 \rightarrow & 0 \\ i,j > 0 \wedge a_i = b_j \rightarrow & F[i-1,j-1]+1 \\ i,j > 0 \wedge a_i \neq b_j \rightarrow & \max\{F[i-1,j], F[i,j-1]\} \end{cases} \tag{2.11}$$

The goal is to find $F[n,m]$. One example of the computation of the table for a simple problem is presented in Figure 2.4. Equation 2.11 presents a monadic formulation since the composition function involves only one term. It is nonserial due to the case when both symbols match. Clearly, the table is $n \times m$ and the dependencies are fixed.

2.4.4 Edit Distance

The edit distance problem is another member of the string matching class. In this problem instead of computing the longest match, it calculates the cost of transforming one string to another one.

Let $T = (t_1, t_2, \ldots, t_i, \ldots, t_n)$ and $R = (r_1, r_2, \ldots, r_j, \ldots, r_m)$ be two strings to compare. We are interested in the cost of transforming string R to string T. We can apply successive comparisons

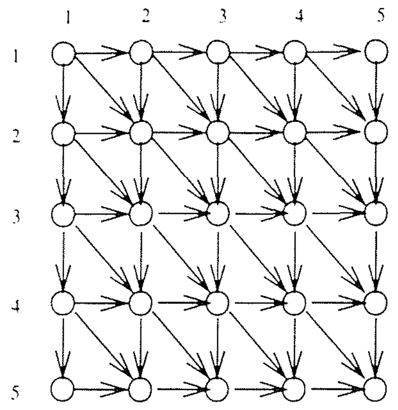


Figure 2.5: Dependencies in the Edit Distance Problem.

between elements of R and elements of T. When a mismatch occurs, we must consider the possibility of replacing a character of T for one of R, of inserting a character of T, or deleting a character of R. This problem can be well defined by the following recurrence relation stating a dynamic programming problem:

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + d(t_i, r_j) \\ D(i-1,j) + K_a \\ D(i,j-1) + K_o \end{cases} \tag{2.12}$$

with the initial conditions:

$$\begin{aligned} D(0,0) &= 0 \\ D(i,0) &= D(i-1,0) + K_a \qquad \text{for } 1 \leq i \leq n \\ D(0,j) &= D(0,j-1) + K_o \qquad \text{for } 1 \leq j \leq m \end{aligned}$$

where $d(t_i, r_j)$ represents the cost of replacing r_j by t_i, K_a the cost of adding t_i, and K_o the cost of omitting r_j. Note that in this formulation, the costs of insertion and suppression are constant, independent from the specific characters. In a typical application, like spelling correction, this calculation has to be repeated a lot of times since the same test string must be compared to many reference strings (a full dictionary for example). Therefore the amount of computation can be very large and prohibits the use of a conventional computer. In Figure 2.5, the dependencies between elements in the table are shown. For the same reasons as the previous subsection problem, this is monadic-nonserial with constant dependencies in a n^2 table.

2.4.5 Gap Problem

The gap problem is a generalization of the edit distance problem when costs of insertions and suppressions are variable and when allowing gaps of insertions and deletions. It can be stated as follows. Given w, w', s_{ij}, and $D[0,0]$, compute

$$D[i,j] = \min \begin{cases} D[i-1,j-1] + s_{ij} \\ \min_{0 \leq q < j}\{D[i,q] + w(q,j)\} \\ \min_{0 \leq p < i}\{D[p,j] + w'(p,i)\} \end{cases} \quad \text{for } \begin{matrix} 0 \leq i \leq m \\ 0 \leq j \leq n \end{matrix} \tag{2.13}$$

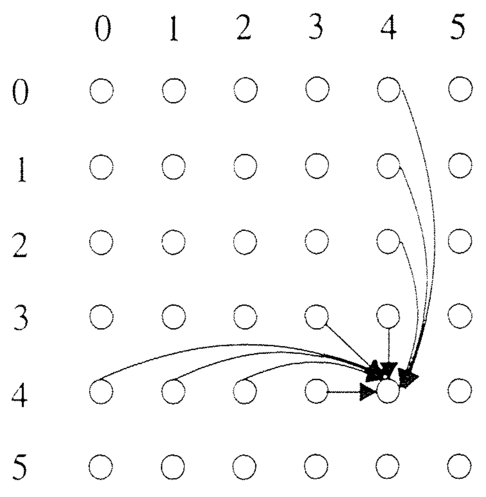


Figure 2.6: Dependencies of the gap problem.

The gap problem arises in molecular biology, geology, and speech recognition [33]. From Figure 2.6 we can observe that this problem describes a two dimensional table where a single subproblem depends on its northwest neighbor and all previous problems along the same row and the same column.

It is easy to see that the computational effort to solve the entry (i, j) takes $O(i + j)$ time. Therefore, a sequential algorithm to compute the problem takes $O(n^3)$ time. It can proceed along rows, columns, or diagonals. It is a monadic problem. It is nonserial since Equation 2.13 involves several subproblems on the right hand side. For the same reason, its dependencies are nonconstant $(O(n))$. It has a n^2 table.

2.4.6 RNA Problem

The RNA problem is the generalization of the two string matching problem. Here the cost function allows insertions, suppressions, and loops with variable cost. Given w and $D[i, 0]$ and $D[0, j]$ for $0 \leq i, j \leq n$, compute

$$D[i,j] = \min_{0 \leq p < i \ \ 0 \leq q < j} \{D[p,q] + w(p,q,i,j)\} \qquad \text{for } 1 \leq i,j \leq n \tag{2.14}$$

This problem has been used to compute the secondary structure of RNA without multiple loops. As can be seen from Figure 2.7, each subproblem depends on all the north-west subproblems. Therefore, the computational effort to solve the entry (i, j) takes $O(i \times j)$ time. Given that we have a two-dimensional table, a sequential algorithm to compute the problem takes $O(n^4)$ time. In a similar way to the Gap Problem, the sequential algorithm can proceed along rows, columns, or diagonals.

The RNA problem refers a monadic-nonserial problem in which the dependencies are variable $(O(n^2)$ and describes a two dimensional table.

2.4.7 Least Weight Subsequence

Given a real-valued function w and $D[0]$, compute

$$D[j] = \min_{0 \leq i < j} \{D[i] + w(i,j)\} \quad \text{for} \quad 1 \leq j \leq n \tag{2.15}$$

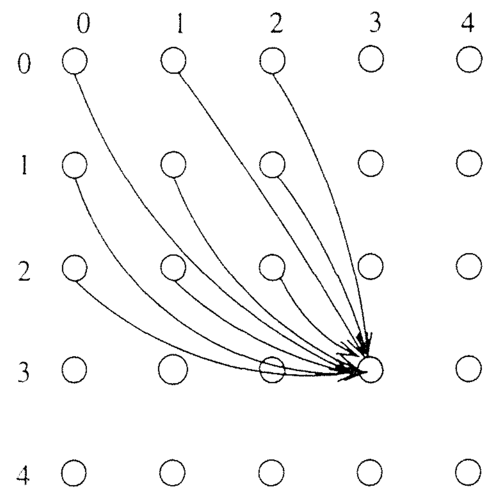


Figure 2.7: Dependencies of the RNA problem.

This problem was called the least weight subsequence problem by Hirschberg and Larmore [42]. Its applications include an optimum paragraph formation problem and the problem of finding a minimum height B-tree.

The subproblem dependencies can be visualized in the Figure 2.8. In Figure 2.8a a two dimensional view of the problem can be observed. An upper triangular table is constructed where the $D[i]$'s values are produced along the main diagonal. Also, a graph representation can be generated for this problem. A directed graph G is built from recurrence 2.15. The vertices of the graph are $0, 1, \cdots, n$. The edges (i, j) for all $i < j$, and edge (i, j) has cost $w(i, j)$. If $f(j)$ is the shortest path from 0 to j in graph G, then, it can be shown that $D[i] = f(i)$. The directed graph view can be observed in Figure 2.8b. It can be shown that the problem stated by recurrence 2.15 is similar to the problem of finding the shortest paths in G from 0 to all vertices.

Recurrence 2.15 presents a monadic-nonserial problem with a variable number of dependencies in a one-dimensional table.

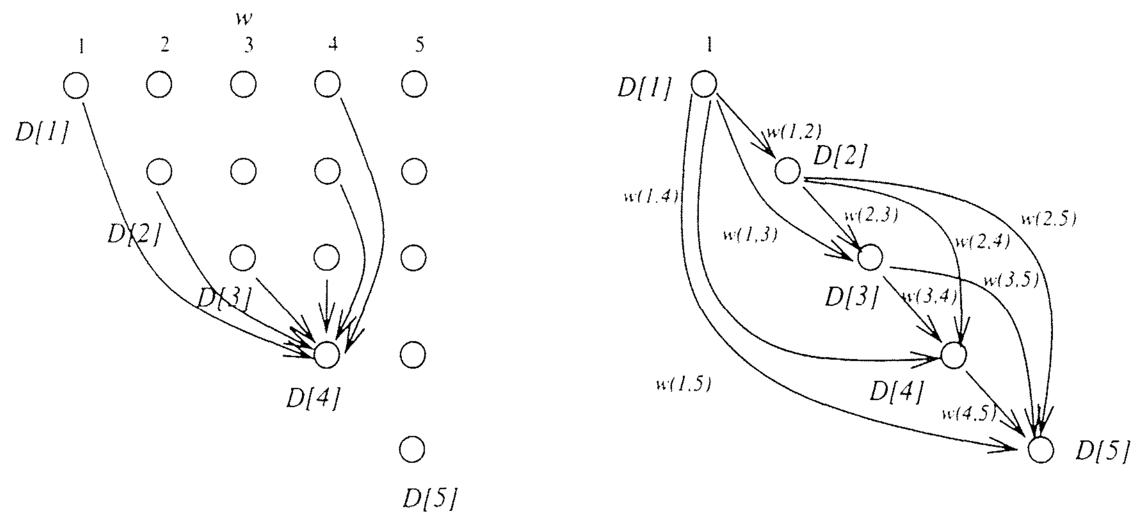
2.4.8 All-Pairs Shortest Paths

The all-pairs shortest paths problem belongs to a more general class of problems known as algebraic path problems. Some additional examples of problems belonging to this class are the transitive closure problem, the matrix inversion problem and the generation of regular languages. All problems belonging to this class share a common formulation. Variation among them depends on the semi-ring in which a specific problem is defined.

We will review here only the all-pairs shortest path problem. We should be aware that all related problems share the same formulation. Given a weighted, directed graph $G = \langle V, E\rangle$ with weight function $w : E \to R$ mapping edges to real valued weights, we are interested in finding a shortest path from u to v for every pair of vertices u and v. An example of this problem can be observed in Figure 2.9.

The well known Floyd's algorithm to solve this problem is based on the following dynamic programming formulation. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \cdots, k\}$. $d_{ij}^{(k)}$ can be computed from the following recurrence equation.

$$d_{ij}^{(k)} = \begin{cases} k = 0 \to & w_{ij} \\ k \geq 1 \to & \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \end{cases} \tag{2.16}$$



a) b)

Figure 2.8: The least weight subsequence problem.

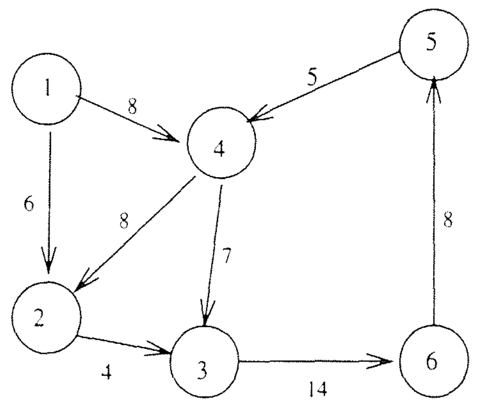
The matrix $D^{(n)} = d_{ij}^{(n)}$ represents the solution to the problem, where n is the number of vertices in the graph. This formulation belongs to the polyadic-serial class because its composition functions involves more than one recursive term, but they are only on the previous level subproblems. It exhibits a n^3 table involving constant dependencies.

2.4.9 Optimal Matrix Parenthesization

Consider the evaluation of the product of n matrices $A_1, A_2, \cdots, A_n$, where each A_i is a matrix with r_{i-1} rows and r_i columns. The order in which the matrices are multiplied together can have a significant effect on the total number of operations required to evaluate the product. Trying all possible orderings in which to evaluate the product of n matrices, so as to minimize the number of operations, is an exponential process which is impractical when n is moderately large. However, dynamic programming provides an $O(n^3)$ sequential algorithm. Let $m(i,j)$ be the cost of multiplying the matrices $A_i, A_{i+1}, \cdots, A_j$. The dynamic programming paradigm constructs the solution to this problem based on the solutions to its subproblems. To compute $m(i,j)$ computes the minimum value of all possible parenthesization between A_i and A_j. This approach gives rise to the following recurrence equation for the parenthesization problem:

$$m(i,j) = \begin{cases} 1 \leq i < j \leq n \rightarrow & \min\limits_{i \leq k < j} \{m(i,k) + m(k+1,j) + r_{i-1}r_k r_j\} \\ j = i, 0 \leq i \leq n \rightarrow & 0 \end{cases} \tag{2.17}$$

Given Eq. (2.17) the problem reduces to finding the value for $m(1,n)$. This problem has been heavily studied. One important aspect of this problem is the nature of interactions between subproblems for solving an specific problem. As can be seen in Figure 5, in order to solve a problem corresponding to the table's entry (i,j) we require all the problems in previous columns to be on the same row and all the problems in down rows to be in the same column. Furthermore, considering Eq. (2.17) they are required to be in a pairwise fashion as shown in Figure 2.10.



$$A = \begin{bmatrix} 0 & 6 & \infty & 8 & \infty & \infty \\ \infty & 0 & 4 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty & 14 \\ \infty & 8 & 7 & 0 & \infty & \infty \\ \infty & \infty & \infty & 5 & 0 & \infty \\ \infty & \infty & \infty & \infty & 8 & 0 \end{bmatrix} \qquad A^{+} = \begin{bmatrix} 0 & 6 & 10 & 8 & 32 & 24 \\ \infty & 0 & 4 & 31 & 26 & 18 \\ \infty & 35 & 0 & 27 & 22 & 14 \\ \infty & 8 & 7 & 0 & 29 & 21 \\ \infty & 13 & 12 & 5 & 0 & 26 \\ \infty & 21 & 20 & 13 & 8 & 0 \end{bmatrix}$$

Figure 2.9: An example of the all-pairs shortest paths problem.

2.5 Dynamic Programming Remarks

From the set of examples we have presented we can see that dynamic programming is basically a decomposition strategy. The general idea is to solve simple subproblems and progressively solve larger problems composed of already solved problems. The computational effort to solve a single subproblem is low, however, considering the dependencies each subproblem has, it seems to lead to inherently sequential algorithms. Other approaches have been followed to overcome with the sequential nature of DP algorithms. Parallel processing and hardware implementation are alternatives. Particularly, hardware level implementation have been followed for several reasons. First, as in the string matching problem, it is common to solve many instances of the problem which requires fast execution. Second, the arithmetic involved in several problems is not so complex. From the examples we have reviewed we can observe that in many cases integer arithmetic with low precision is enough. Finally, being able to re-organize the recursive formulations in such a way that we obtain local and regular dependencies we can implement them in FPGA circuitry. Techniques to achieve that will be discussed in chapter 5.

2.6 Conclusions

We have presented the general formulation of dynamic programming algorithms. We showed how the decomposition of a problem into subproblems can lead to a DP formulation. We reviewed the classifications of DP problems. From a set of examples, we have made some observations about efficient implementations of DP formulations. It has been believed that dynamic programming is adequate for sequential computers and not appropriate for parallel machines. In the next chapter we will show how to exploit parallelism in dynamic programming formulations.

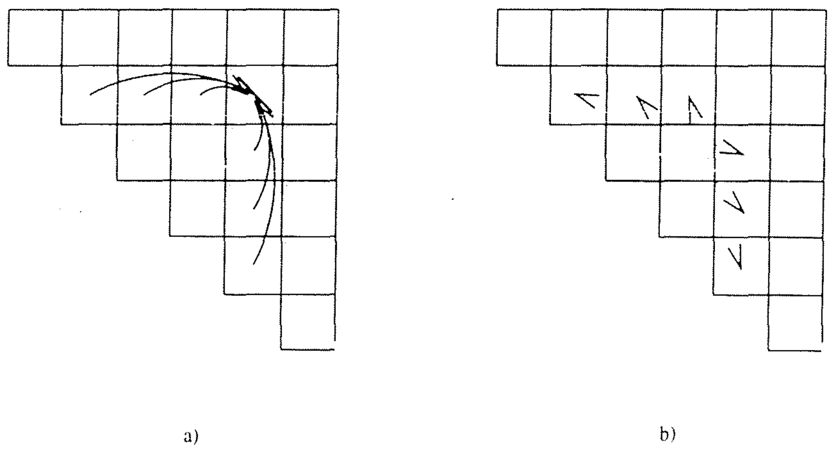


Figure 2.10: An instance of the optimal matrix parenthesization problem.

Chapter 3

Data-Parallel Dynamic Programming

In this chapter we describe general strategies for implementing parallel dynamic programming algorithms. We start by making some observations about exploiting parallelism in DP algorithms. Then, we present the C* programming model, our target programming language. Then we discuss a straightforward approach for implementing DP algorithms in C*. Finally, we present a block decomposition approach to implement DP algorithms with a variable number of dependencies.

3.1 General Guidelines

In the last chapter we presented a general classification of DP algorithms. In this section we discuss general guidelines to exploit parallelism of DP algorithms according to their formulation.

- That a problem is serial or nonserial affects the way in which we build the DP table. In a serial problem, only the current and the previous level subproblems need to be stored. In a nonserial problem, solved subproblems on all previous levels are required. In general, all subproblems must be kept for future references.

- The only source of parallelism for serial problems is among problems in the same level. All of them can be computed in parallel, but no problems belonging to different levels can be computed concurrently.

- The computation of nonserial formulations allows us to overlap computing problems from different levels. This can be a great source of parallelism which can help us derive efficient practical implementations. However, we need to derive a scheduling policy to establish the order in which the entries of the table are going to be filled.

- For systolic processing, nonserial problems must be serialized. The way in which the serialization is done greatly impacts the performance of the resulting systolic array. For example, Guibas and colleagues [39] provided a clever mechanism for the OMP problem. Chen [18] discussed a more systematic approach for that serialization.

- Whether a problem is monadic or polyadic influences the way in which the solution of a specific problem is built (the second stage in the algorithmic point of view). Solutions of monadic problems are built in serial fashion. Solutions of polyadic problems can be built by searching AND trees.

- The number of dependencies is important to generate parallel programs. That a problem depends in a variable number of subproblems implies that we need to serialize the optimization operation (minimum or maximum). For serial problems, the necessary serialization cannot be

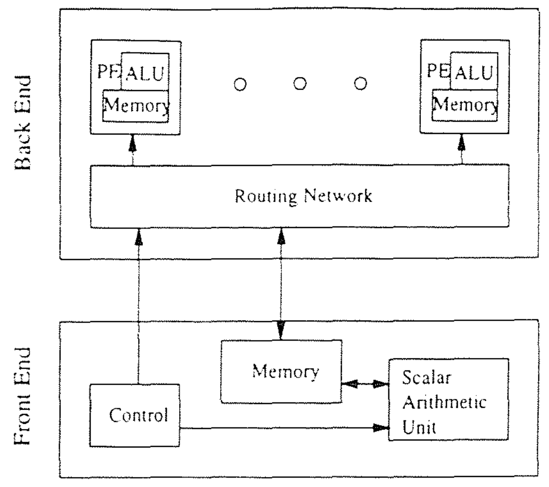


Figure 3.1: The C* Programming Model.

done until the preceding level subproblems are computed. That implies necessarily a sequential algorithm. For nonserial problems, the serialization can be performed by overlapping different problems in different levels.

- The dimensionality of the table provides us a clue to the dimensionality of the processor array to compute the algorithm. Using one processor per entry in the table gives us exactly the number of dimensions of the processor array. Through a combination of scheduling and mapping transformations we can reduce the dimensionality. For example, the longest common subsequence problem has a 2D table. The preceding naive approach would underuse the processors because only a limited number of them are active at the same time. Using a hyperplane method to schedule the filling of the table, we obtain a 1D processor array.

We believe that three important features must be considered to derive parallel implementations: the size of the table, the seriability of a recurrence formulation and the number of dependencies. Based on that, we can find several algorithms for different problems.

3.2 C* Programming Model

Our target language is the C* programming language. The C* programming model is based on SIMD computations consisting of a front-end uniprocessor attached to an adaptable back-end parallel processor as it appears in Figure 3.1. The sequential portion of the C* program (consisting of sequential code) is executed on the front-end. The parallel portion of the C* program is executed on the back-end. A global name space is used to reference variables in the front-end by processors in the back-end. C* executes programs using a single instruction stream in a synchronous way.

C* uses the `shape` keyword to describe the size and shape of parallel data. A virtual processor per position, cell in the shape, is considered. A parallel variable is declared by tagging it with a shape. The `with` keyword serves to choose the shape to be used for parallel operations, and the `where` statement restricts operations to some positions (virtual processors).

3.3 A Straightforward Approach for Parallel Dynamic Programming

In this section we describe a straightforward approach to exploit parallelism in implementing DP algorithms. We consider a data-parallel approach consisting of an array of virtual processors with the same shape as the DP table. All processors perform the same computation. In this naive approach we use a virtual processor per table entry whose role is to compute the value of the entry. However, for serial problems we do not need as many processors as entries in the table because, as we mentioned before, we need only the current and the preceding level of subproblems.

3.3.1 Data-parallel loops of DP problems

In the following sections we will examine specific cases for important parallel formulations.

Serial DP problems

In this section we are considering the case of loops used for serial, monadic or polyadic, formulations of dynamic programming problems which we call *serial data-parallel loops.* Their structure is reduced to the following scheme:

```
shape [ ] ... [ ]S;
...:S m;

. . .
with( S ) {          /* Current shape operation */

  ... /* initialization */

  for( k = 1; k <= N; k++ ) {
    m = ...m...
    ...                /* References and assignments to parallel vars. */
  }
}
```

The implementation consists of a sequential loop with references to parallel variables in its body. It is supposed that a level of subproblems is represented by the parallel variable m. The number of iterations of the loop represents the number of levels in the DP table. All virtual processors perform the parallel operations in which m is involved. That means that in every iteration of the loop, the values of m are updated with new values. Because of the SIMD model of C*, we can avoid explicitly storing the previous level, given that any reference in the right hand side of an assignment is done before the assignment is actually done.

One example of this type of code is obtained from the all-pairs shortest path problem. In Figure 3.2 we can observe the problem's dependencies which can be complicated to understand all of them, however, looking only at the k-dimension we easily recognize the serial nature of this problem. The problem can be computed from the following code:

```
shape [N+1][N+1] Mesh;
int:Mesh M;

with( Mesh )
  for( k = 1; k <= N; k++ )
      M = (M <? ([.][k]M + [k][.]M);
```

in which the operator <? selects the minimum of the two arguments and [.] refers to virtual processor on the same row or column according the position it appears. Note that the composition function is polyadic because it refers to two different subproblems.

Nonserial DP problems

Let us examine first the structure of nonserial problems applying an obvious scheme to serialize operations. We do not start to compute any problem unless all problems in which it depends on

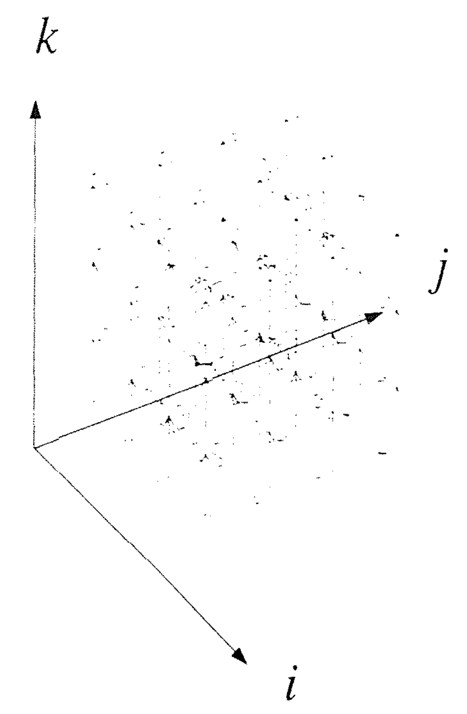


Figure 3.2: Organization of the shortest path problem by levels.

have been already computed. That implies a sequential organization of subproblems, where not all of them are ready at any time. The structure of the naive implementation would be the following:

```
shape [ ] ... [ ]S;
...:S m;

. . .
with( S ) {        /* Current shape operation */

  ... /* initialization */

  for( k = 1; k <= N; k++ )
    where( P( ... ) ) {  /* Selection of active positions */
      m = ...m...
      ...              /* References and assignments to parallel vars. */
    }
}
```

The where keyword is used here to select those virtual processes for which the predicate P is true. This predicate establishes a scheduling policy which activates or deactivates virtual processors. In section 3.4, we will see how to determine that predicate. For the moment let us consider the case where we have a monadic or polyadic formulation in which the code enclosed by the where statement consists only of a fixed number of instructions. For example, we obtain the following code for the longest common subsequence problem.

```
shape [N+1][M+1] Table;
shape [] String;

int:Table F, ii, jj;
char:String s1, s2;
```

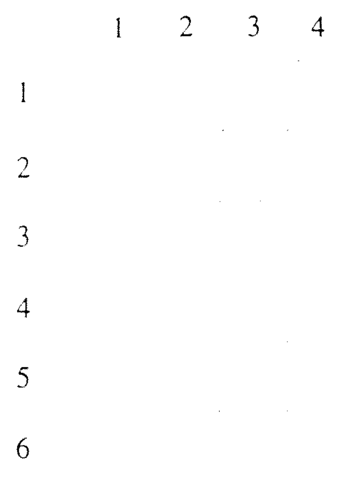


Figure 3.3: The longest common subsequence DP table and dependencies.

```
with( Table } {
  where( pcoord(0) == 0 ) F = 0;
  where( pcoord(1) == 0 ) F = 0;

  for( k = 1; k < M+N; k++ ) {
    where( ((pcoord(0) + pcoord(1)) == k) && (pcoord(0) > 0) && (pcoord(1) > 0) ) {
      ii = pcoord(0) - 1;
      jj = pcoord(1) - 1;
      where( [ii]s1 == [jj]s2 )
        F = [.-1][.-1]F + 1;
      else
        F = ([.-1][.]F >? [.][.-1]F);
    }
  }
}
```

In this case the predicate determines a wavefront for activating the virtual processors as is shown in Figure 3.3. It involves the use of the function pcoord(d) which returns the position that occupies a virtual processor in the coordinate d of the array.

For nonserial problems with a variable number of dependencies the code composing the where statement consists of as many loops as indicated by the order of the dependencies. For example, we obtain the following code for the optimal matrix parenthesization problem.

```
shape [N][N] Mesh;
shape [N+1] Dims;

int:Mesh m, ii, jj, kk;
int:Dims r;

with( Mesh ) {
  where( pcoord(0) = pcoord(1) )
    M = 0;

  for( d = 1; d <= N; k++ ) {
    where( (pcoord(1) - pcoord(0)) == d ) {
```

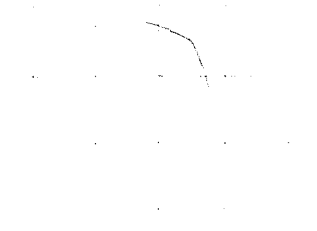


Figure 3.4: Wavefront for the optimal matrix parenthesization problem.

```
          ii = pcoord(0); jj = pcoord(1) + 1; kk = pcoord(0) + 1;
          M = [.][pcoord(0)]M + [.+1][.]M + [ii]r*[kk]r*[jj]r;
          for( k = 1; k < d; k++ ) {
            kk = pcoord(0)+k+1;
            M <? = [.][pcoord(0)+k]M + [.+k+1][.]M + [ii]r*[kk]r*[jj]r;
          }
        }
      }

    }
```

Although a two-dimensional array is declared, only $n(n+1)/2$ processors are really used; one processor per entry in the triangular table. They only are active across diagonals, as shown in Figure 3.4. That is, each processor is active during only one iteration of the outermost loop. Note that the outermost loop's body is itself a loop, because of the $O(n)$ dependencies which every subproblem depends on. Because of that reason this implementation takes $O(n^2)$ time.

3.4 A Practical Approach for Parallel Dynamic Programming

In this section we describe a more practical approach to implement DP algorithms. The naive approach serves to point out the importance of the seriability, the size of the table and the nature of dependencies in a problem. As Hatcher and Quinn [41] suggested, the use of virtual processors makes programs simpler and shorter. However, in nonserial dynamic programming algorithms, virtual processor emulation adds a great deal of overhead to the code generated. That is due to the fact that only a subset of virtual processors are active at any one time. Our goal is to derive an *allocation function* to map all the entries of the table to a limited number of processors and a *timing function* to decide which entries are going to be filled at any time.

Let us consider a DP table, M, defined at every index point, z, of the domain $\mathcal{D}$. Timing functions must satisfy the *causality property* [30] which says that if index point z depends on z' then

$$t(z) > t(z')$$

In other words, the time at which a computation is done must be greater than the times at which all its dependence points are computed. Allocation functions determine the set of virtual processors that will fill the DP table. For every entry of the table, z, the virtual processor identified by $a(z)$ will be in charge of filling the entry z. Allocation functions must be conflict free, i.e., two different entries scheduled at the same time have to be filled for different processors. That is,

$$\forall z, u \in \mathcal{D} : \qquad \text{if } \ t(z) = t(u), \ \text{ then } \ a(z) \neq a(u)$$

We will defer the discussion about determining timing and allocation functions until Chapter 4. Now we will illustrate two cases: when we have fixed constant dependencies and when we have non-fixed dependencies.

3.4.1 Uniform Dependencies

Let us consider the longest common subsequence problem that we introduced in Chapter 2. This problem exhibits the fixed uniform dependencies, $delta_i$, which means that point z depends on $z - \delta$

$$\delta_1 = [1,0]^T, \qquad \delta_2 = [0,1]^T, \qquad \text{and} \qquad \delta_3 = [1,1]^T$$

It can be easily seen that by scheduling entry (i,j) at time $i + j$ we can solve the causality property of timing functions. That timing function is represented by the scheduling vector $[1,1]^T$.

Now, if we always choose the shortest string to represent the j-dimension, then we can project the DP table onto the $[01]^T$ direction leading us to avoid scheduling conflicts. That means that entry (i,j) is executed by processor j.

Representing this transformation in a matrix form we can obtain the direct and inverse transformation as:

$$\begin{bmatrix} t \\ x \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}, \qquad \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t \\ x \end{bmatrix}$$

This mapping generates a linear processor array with as many processors as columns as the table of the longest common subsequence problem has. The dependencies $[1,0]^T$ and $[1,1]^T$ are mapped to the previous processor and the dependencies along the direction $[0,1]^T$ are mapped onto the same processor. So, every processor should be able to maintain the last two values in addition to the previous one. The code generated using a linear array is the following:

```
shape [M+1] Table;
shape [] String;

int:Table Fc, Fold1, Fold2, ii, jj;
char:String s1, s2;

with( Table } {

  where( pcoord(0) == 0 ) Fc = 0;
  Fc = 0;


  for( k = 0; k < M+N; k++ ) {
    where( (k - N) < pcoord(0) && pcoord(0) < k )  {
      Fold2 = Fold1;
      Fold1 = Fc;
      ii = k - pcoord(0);
      jj = pcoord(0) - 1;
      where( [ii]s1 == [jj]s2 )
        Fc = [.-1]Fold2 + 1;
        else
          Fc = [.-1]Foldl >? Foldl;
      }
    }
  }
```

3.4.2 Non-uniform Dependencies

Non-uniform dependencies appear in DP problems in the cases of the all-pairs shortest paths problem and all problems with more than $O(1)$ dependencies. When we have nonuniform dependencies, two approaches can be considered to deal with this kind of problem: using dependency direction vectors or uniformizing the recurrence formulation. For the moment, we are going to apply the first technique to the OMP problem. The uniformization approach will be discussed in Chapter 6.

Dependency Direction Vectors

We can suppress the exact dependency vector with direction dependency vectors just as in the same way parallelizing compilers use dependence direction instead of exact dependency vectors [81]. That means that we are assuming that a problem depends on all subproblems lying along a direction vector, including the previous level problems. Hence, we are implicitly transforming a nonserial problem to a serial one. Then, we can apply the technique described in the preceding section to determine a timing and an allocation function. This approach does not exploit the overlapping of different level subproblems, so it leads inefficient implementations.

Let us examine the application of this approach to the optimal matrix parenthesization (OMP) problem which we introduced in Chapter 2. In the OMP problem, every subproblem depends on all subproblems lying in the same row in preceding columns and the same column in subsequent rows (see Figure 3.4). The dependence direction vectors are $[1,0]^T$ and $[0,-1]^T$. A timing function satisfying the causality property can proceed along diagonals that can be represented by the following function:

$$t(i,j) = -i + j$$

To avoid scheduling conflicts, we can project the table along rows using the following allocation function

$$a(i,j) = i$$

The direct and inverse transformation are given in matrix form as:

$$\begin{bmatrix} t \\ x \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}, \qquad \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} t \\ x \end{bmatrix}$$

That mapping yields a linear array of processors. After applying the allocation function, the dependency vector $[-1,0]^T$ implies a communication between a processor and the next one in the array. The dependency vector $[0,1]^T$ is mapped onto the same processor so it implies that every processor stores the results generated along the same row. At every diagonal step we accumulate the results generated in the same row and we pass them to the next processor.

The following code uses the timing and allocation functions generated before.

```
shape [MAXN] Rows;

struct col_results {
    int value[MAXN];
```

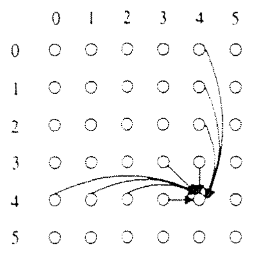


Figure 3.5: Dependencies of the gap problem.

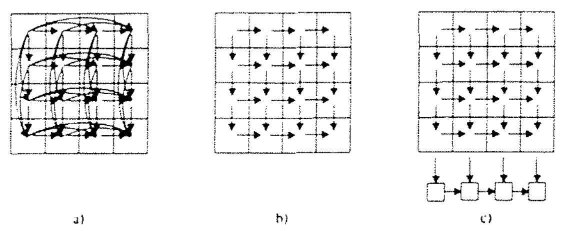


Figure 3.6: Block decomposition of the Gap Problem.

$$D[i,j] = \min \begin{cases} D[i-1,j-1] + s_{ij} \\ \min_{0 \le q < j}\{D[i,q] + w(q,j)\} & \text{for } \begin{matrix} 0 \le i \le m \\ 0 \le j \le n \end{matrix} \\ \min_{0 \le p < i}\{D[p,j] + w'(p,i)\} \end{cases} \tag{3.1}$$

From Figure 3.5 we can observe that this problem describes a two dimensional table where a single subproblem depends on its northwest neighbor and all previous problems along the same row and the same column.

It is easy to see that the computational effort to solve the entry (i,j) takes $O(i+j)$ time. Therefore, a sequential algorithm to compute the problem takes $O(n^3)$ time. It is easy to see that if we process the table along diagonals we can solve in parallel all the problems along the same diagonal.

3.5.2 Block Decomposition for the Gap Problem

We can observe that as soon as a single subproblem is computed it can be used to partially solve the subsequent problems in the same column and in the same row. However, that approach would lead to a great number of communications in a multicomputer (a parallel computer with a distributed memory). Given that communications costs are at least one order greater than computations costs, we need to balance their effects in a parallel algorithm.

A common approach to balance communications with computations is to partition the table into blocks in such a way that communications occur only when complete blocks are solved (see Figure 3.6a). Considering now dependencies between blocks, the partitioned table would show the same shape as the original table. Taking into account that an acceptable schedule proceeds along diagonals, we can avoid long communications by mixing messages and communicating only neighbor blocks as shown in Figure 3.6b. Note that mapping a single column of blocks to a single processor as long as we use the diagonal schedule would provide a valid parallel algorithm (Figure 3.6c).

Another way to improve speedup is to overlap computations with communications. Although the limitations of such an approach have been reported by Quinn and Hatcher [66], we can use it here to slightly improve the speedup of the parallel algorithm. In Figure 3.7, we can observe the execution

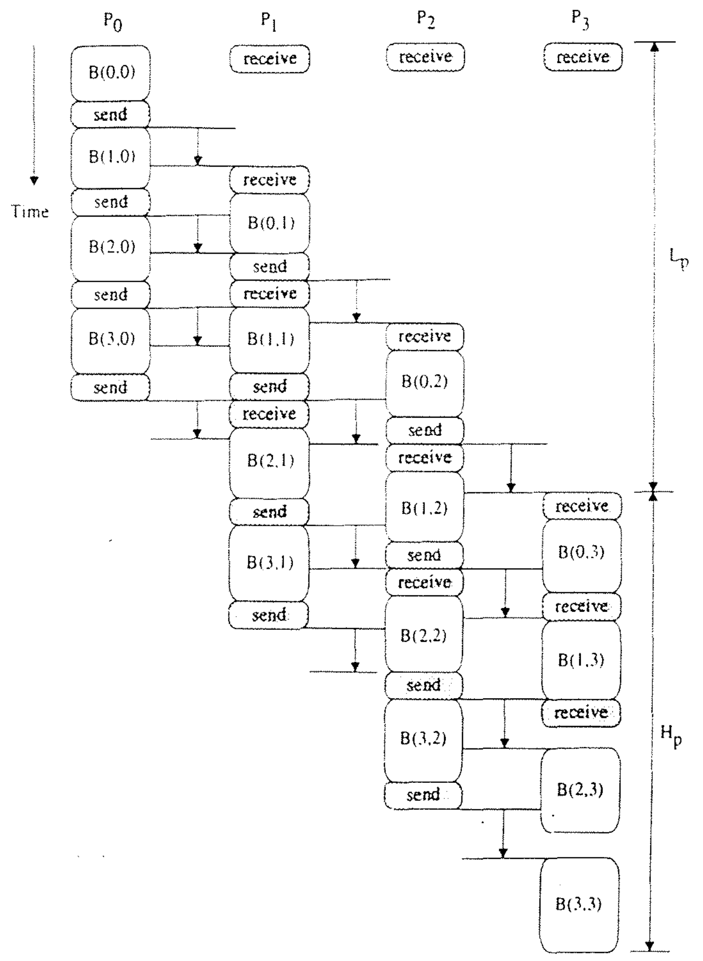


Figure 3.7: The parallel execution of the block decomposition scheme for the Gap Problem allowing overlap of computations with communications.

of this scheme when the communications do not take more time than computing a single block plus executing the primitive for receiving a message. It is expected that this is the real situation for most of the cases because it is unexpected that the number of processors increases with the size of the problem. Furthermore, to exploit more parallelism, we can reduce the size of blocks by making more horizontal divisions in our decomposition scheme. To finish this discussion, this problem belongs to the class of problems that Quinn and Hatcher discussed as having a computational complexity greater than the communication complexity. It is clear that the communication requirements are $O(n^2)$ while the computation ones are $O(n^3)$

In summary, each processor performs the following pseudocode:

```
if( pid > 1 )   receive();
 for( i = 1; i <= Number of blocks; i++ ) {
   if( pid > 1 ) {
     while( not received message )  ;
     if( i < Number of blocks )
      receive();     /* next block */
   }
   /* Process block i */
   if( pid < Number of blocks )
     send();         /* received message+block i to next processor */
}
```

Two important characteristics should be realized to implement the block decomposition for the gap problem. First, as we proceed solving problems, it takes more time to solve the remaining ones. Second, the size of communications increases from processor to processor proportionally to the macrocolumn position that each processor computes. So let us now develop a model to predict the expected execution time of the algorithm.

3.5.3 Homogeneous Block Decomposition

To allow overlap of computation with communication, we consider a non-blocking communication model where the time to send a message of size n is given by $L(n) + T(n)$. $L(n)$ is the message latency which is expressed by the function $\lambda + \beta n$, where λ is the constant term representing the time required to handle the call to send or receive, β is inversely proportional to the speed at which the system can buffer or unbuffer the message. $T(n)$ is the time spent to transmit a message of size n by the communication network. Usually, it is a linear function χn where χ is the inverse of the fixed bandwidth of the routing network. We assume also that the time to compute a single selection operation of the optimization process takes τ_a time.

A standard compiler would divide the DP table equally among processors, so if we have as many macrocolumns as processors, we are assuming that a table of size $n \times m$ is divided into $b \times p$ blocks of size $n/b \times m/p$.

From Figure 3.7 we can observe that the overall execution time of this parallel algorithm is dominated by the latency time of the last processor and its own execution time. The execution time for this homogeneous block decomposition is given by

$$T_H(n, p, b) = L_{H_p} + H_{H_p}$$

where L_p means the latency time of the last processor and H_p means the throughput of the last processor. Both components involve a computation and a communication part. The computation part is determined by the blocks involved in computation, and the communication time indicates the communication primitives involved. We can express,

$$L_{H_p} = L_{H_{comp_p}} + L_{H_{comm_p}}$$

and

$$H_{H_p} = H_{H_{comp_p}} + H_{H_{comm_p}}$$

For the latency component we have to compute the first $(p-1)n/p$ columns of the first n/b rows. Therefore, the computation part of the p processor latency is:

$$\begin{aligned} L_{H_{comp_p}} &= \tau_a \left(\sum_{i=1}^{N/b} \sum_{j=1}^{(p-1)N/p} i + j \right) \\ &= \tau_a \left(\frac{(p-1)^2 b + (p-1)p}{2p^2b^2} n^3 + \frac{p-1}{pb} n^2 \right) \end{aligned} \tag{3.2}$$

Regarding the communication part, $(p-1)$ messages are sent, causing $p-1$ calls to send. The size of each message is $q\frac{n}{b}\frac{n}{p}$, with $1 \le q < p$. There are done $p-2$ receiving calls during the computation and communication overlapping processes. The communication part of the latency time is expressed as follows:

$$\begin{aligned} L_{comm_p} &= \sum_{i=1}^{p-1} \lambda + \beta i \cdot \frac{n}{b}\frac{n}{p} + \chi i \cdot \frac{n}{b}\frac{n}{p} \quad + (p-2)\lambda \\ &= (2p-3)\lambda + \frac{p-1}{2b}(\beta + \chi) n^2 \end{aligned} \tag{3.3}$$

Now let us consider the throughput of the last processor. It computes the last n/p columns of the table. Its computation time is:

$$\begin{aligned} H_{H_{comp_p}} &= \tau_a \sum_{i=1}^{N} \sum_{(p-1)N/p+1}^{N} i + j \\ &= \tau_a \left(\frac{3(p-1)}{2p^2} n^3 + \frac{1}{p} n^2 \right) \end{aligned} \tag{3.4}$$

Let us assume that messages received by the last processor are short enough to completely overlap their transmission time with the computation of a block. Hence only $b-1$ calls to receive are required. The time spent in communication by the last processor is:

$$H_{H_{comm_p}} = (b-1)\lambda \tag{3.5}$$

Adding together Eqs. 3.2 - 3.5, we obtain the execution time for the homogeneous block decomposition scheme:

$$\begin{aligned} T_{HBD} &= (2p+b-4)\lambda + \frac{p-1}{2b}(\beta+\chi)n^2 \\ &+ \tau_a \left[\left(\frac{(p-1)^2 b + (p-1)p}{2p^2 b^2} + \frac{3(p-1)}{2p^2} \right) n^3 + \left(\frac{p-1}{2b} + \frac{1}{p} \right) n^2 \right] \end{aligned} \tag{3.6}$$

As can be seen from Equation 3.6, the execution time is dominated by the computation time of the last processor. It is important to see that this homogeneous block decomposition does not evenly distribute the computational work among processors because the hardest n/p columns are assigned to processor p.

3.5.4 Non-Homogeneous Block Decomposition

Let us now develop a non-homogeneous block decomposition scheme where the computation work is evenly distributed among processors. Let us assume that we have a $N \times M$ table to distribute among p processors. The total number of operations for the gap problem is proportional to

$$\sum_{i=1}^{N} \sum_{j=1}^{M} i + j = \frac{N}{2} M^2 + \left(\frac{N^2}{2} + N \right) M$$

We would like that each processor would compute

$$\frac{N^3 + N^2}{p}$$

amount of computational work. In other words, let us assume that the first K columns are computed by $p-1$ processors. We need that

$$\sum_{i=1}^{N} \sum_{j=1}^{K} i + j = \frac{p-1}{p} \left(\frac{N}{2} M^2 + \left(\frac{N^2}{2} + N \right) M \right) \tag{3.7}$$

$$\left(\frac{N}{2} K^2 + \left(\frac{N^2}{2} + N \right) \right) K = \frac{p-1}{p} \left(\frac{N}{2} M^2 + \left(\frac{N^2}{2} + N \right) M \right) \tag{3.8}$$

It can be seen that by choosing the positive solution of the equation:

$$NK^2 + (N^2 + 2N)K - \frac{p-1}{p}\left(NM^2 + (N^2+2N)M\right) = 0$$

we can find the value for K that separates the work for the last processor from the others. Recursively applying the equation and reducing the number of processors we can find the sequence $K_1, K_2, \cdots, K_{p-1}$ of columns which partitions the dynamic programming table of the gap problem letting each processor perform the same amount of computational work.

The model to predict the performance of the non-homogeneous block decomposition has similar components to the homogeneous one.

The execution time for the non-homogeneous block decomposition is given by

$$T_N(n,p,b) = L_{N_p} + H_{N_p}$$

We can express the computation and the communication part of each component in the following way:

$$L_{N_p} = L_{N_{compp_p}} + L_{N_{comm_p}}$$

$$H_{N_p} = H_{N_{compp_p}} + H_{N_{comm_p}}$$

A good approximation for the number of columns assigned to p-1 processors is

$$K = \frac{2p-1}{2p}N$$

The computation time of the latency component can be approximated by

$$\begin{aligned} L_{N_{compp_p}} &= \tau_a \left[\sum_{i=1}^{N/b}\sum_{j=1}^{(2p-1)N/p} i+j\right] \\ &= \tau_a \left[\frac{2p-1}{4pb}\left(\frac{1}{b}+\frac{2p-1}{2p}\right)n^3 + \frac{2p-1}{2pb}n^2\right] \end{aligned} \tag{3.9}$$

Regarding the communication part, $(p-1)$ messages are sent, requiring $p-1$ calls to send. Longer messages are required for the non-homogeneous block decomposition. In the average, the messages are of size $0.6n$. Note that $p-2$ calls to receive are done in the process of overlapping computation and communication. The communication part of the latency time is expressed as follows:

$$L_{N_{comm_p}} = (2p-3)\lambda + 0.6\frac{p-1}{b}(\beta+\chi)n^2 \tag{3.10}$$

According to the decomposition scheme, the computation time of the last processor is:

$$H_{N_{compp_p}} = \tau_a\left(\frac{1}{p}n^3 + \frac{1}{p}n^2\right) \tag{3.11}$$

Finally, assuming again that the computation time is large enough to completely overlap the communication time of the messages which the last processor receives, the time spent in communication by the last processor is:

$$H_{N_{comm_p}} = (b-1)\lambda \tag{3.12}$$

Adding together Eqs. 3.9 - 3.12, we obtain the execution time for the non-homogeneous block decomposition scheme:

$$\begin{aligned} T_{NBD} &= (2p+b-4)\lambda + 0.6\frac{p-1}{b}(\beta+\chi)n^2 \\ &\quad + \tau_a\left[\left(\frac{b+1}{2b^2}+\frac{1}{p}\right)n^3 + \left(\frac{1}{b}+\frac{1}{p}\right)n^2\right] \end{aligned} \tag{3.13}$$

3.5.5 Discussion

From Equation 3.6 and 3.13 we can see that the significant part of the execution time in both schemes is given by the cubic power of the problem size. Although the latency time is greater for the non-homogeneous block decomposition, the difference in computation time is of order 1.5 times more on the homogeneous block decomposition scheme.

In Figure 3.8 we present the execution time vs number of blocks for a problem of size $n = 512$. Parts a) and b) correspond to the Meiko CS-2 system for 2, 4, 8, and 16 processors and parts c) and d) correspond to the CM-5 parallel computer with 2, 4, 8, 16, and 32 processors. Figures 3.8a,c present the execution times for the homogeneous block decomposition, and Figure 3.8b,d presents the same for the non-homogeneous block decomposition scheme. The dashed lines show the observed execution times. The solid lines show the predicted execution times based on the models derived in previous sections. The parameters observed for the Meiko CS-2 were: $\tau_a = 0.9\mu$secs, $\lambda = 30\mu$secs, $\beta = .001\mu$secs/integer, and $\delta = 0.1\mu$secs/integer. The parameters observed for the CM-5 were: $\tau_a = 3.8\mu$secs, $\lambda = 100\mu$secs, $\beta = 0.5\mu$secs/integer, and $\delta = 0.1\mu$secs/integer.

From those figures we realize that after certain number of blocks, the gain in execution time is negligible. By increasing the number of blocks, we reduce the latency of the last processor. However, that is compensated by the greater number of messages that the last processor must receive. In any case, the best execution time was observed when the number of blocks was equal to the size of the problem making blocks of single rows. Andonov and Rajopadhye found similar curves for a homogeneous block decomposition of the knapsack problem [6]. They obtained analytical solutions for the optimal block's size for that problem. Considering all possible cases, they found that the optimal solution is when the number of columns is equal to the number of processors or when the number of rows is one.

In Figure 3.9 we compare the speedup obtained for the homogeneous and non-homogeneous block decomposition schemes for the Meiko CS-2 and the CM-5. The a and c parts show the speedup for a problem of size $n = 512$ and b and d parts show the speedup for $n = 1024$. Again, dashed lines show observed speedup and solid lines show predicted speedup. In both cases, an important improvement can be observed by using the non-homogeneous scheme.

When the problem's size increases, cache and vector unit effects avoid to follow the model $\tau_a(N^3 + N^2)$ for the sequential algorithm. That explains the shift between the predicted and the observed speedup.

3.6 Related Work

Sequential algorithms based on the dynamic programming paradigm have been proposed for individual problems [24, 33, 35, 42, 76]. However, due its large computational requirements, the applicability of DP has been somewhat limited. Different approaches has been proposed to deal with DP complexity.

Galil *et al* derived effective sequential implementations where the cost function associated with some DP problems exhibits properties such as concavity, convexity and sparsity [35, 26, 27]. In [35],

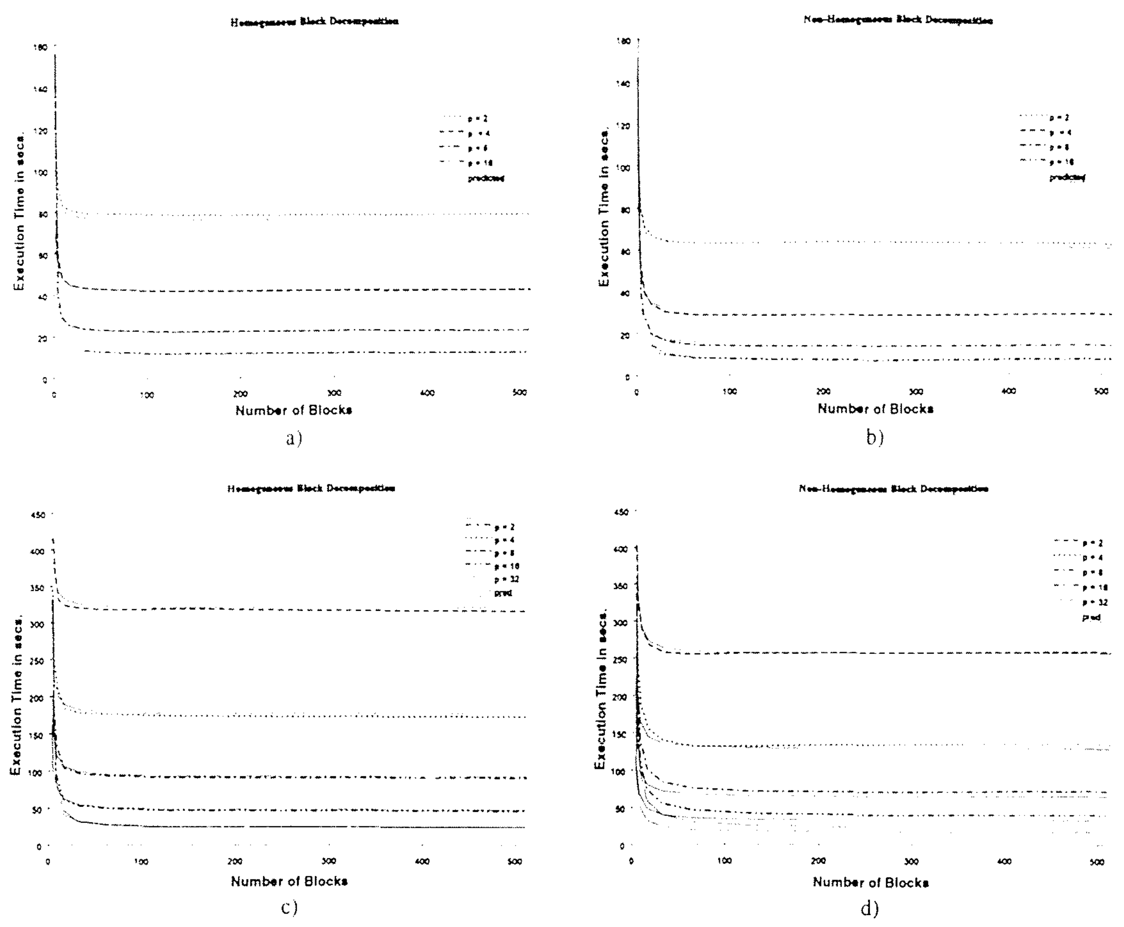


Figure 3.8: Effect of the number of blocks int the execution time on the Meiko CS-2 a y b, and on the CM-5 c and d. a) and b) show homogeneous block decomposition. b) and d) show non-homogeneous block decomposition.

Galil and Park discussed the classification DP problems considering the order of the DP table size and the order of the dependencies to solve a single subproblem.

Guibas *et al* proposed a VLSI algorithm to solve the optimal matrix parenthesization (OMP) problem [39]. Li and Wah [53] proposed a classification of DP problems for parallel processing. They classified DP problems according their recursive formulation to distinguish between serial and nonserial problems, and between monadic and polyadic formulations. They discussed the implementation of systolic algorithms for problems in different classes.

Recently, parallel algorithms based on PRAM models have been proposed for the optimal matrix parenthesization problem (OMP). Bradford developed an algorithm requiring $O(n^6/\log n)$ processors to solve the OMP problem in time $O(\log^2 n)$ [15]. Huang and colleagues [44] modified an algorithm proposed by Rytter [74] to solve the OMP problem in the same time using $O(n^6/\log^5 n)$ processors. Galil and Park proposed an optimal algorithm solving the problem in $O(n^{3/4}\log n)$ time using $O(n^4/\log n)$ processors [34]. These last authors went beyond that problem, considering problems having more than $O(1)$ dependency, that is, problems whose formulations involve a variable number of subproblems. Although efficient PRAM algorithms have been provided, they are not suitable for implementation on actual parallel computers.

Practical parallel implementations have also been derived for individual problems. Kumar *et al* [49] describe practical implementations of some DP problems according the classification previously proposed by Li and Wah. In [46], Karipys and Kumar discussed three different mappings of the

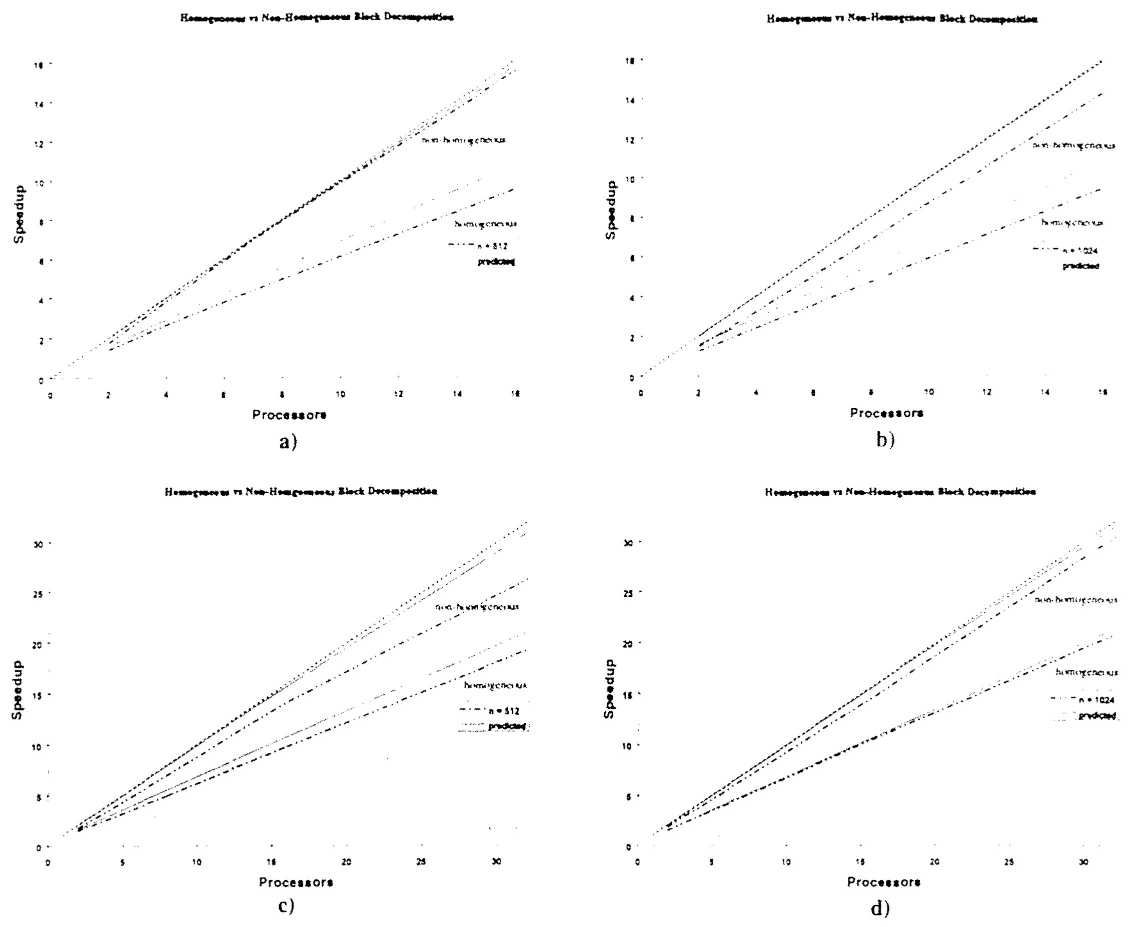


Figure 3.9: Speedup of homogeneous and non-homogeneous decomposition schemes on the Meiko CS-2 on the CM-5. a) and c) n= 512. b) and d) n = 1024.

systolic algorithm for the OMP problem onto multicomputers. The OMP problem has been the favorite DP problem for many researchers because it is isomorphic to a number of problems such as the optimal convex polygon triangulation problem and the optimal binary search tree construction problem [24]. However, little attention has been focused on generating design strategies to build practical parallel implementations for general dynamic programming problems.

3.7 Conclusions

We have discussed the main requirements to generate data-parallel programs for dynamic programming algorithms. We extracted the important characteristics in generating data-parallel implementations. Data-parallelism is quite natural for DP problems because their recurrent formulations establish the same computation for all entries in a DP table.

The seriability, the number of dependencies and the size of the table are useful to generate naive data-parallel programs of DP problems. We presented the general structure of C* programs implementing serial and nonserial DP formulations. The naive approach was further elaborated to generate more effective implementations for nonserial problems. These elaborations are based on overlapping the computation of problems in different levels. The goal is to design a scheduling function for filling the entries of the DP table and an allocation mapping to design a virtual processor array in charge of doing that job.

We applied previous results from the synthesis of systolic arrays and automatic loop parallelization in the process of designing algorithms for DP problems. We proposed the use of dependence direction vectors and the uniformization of the formulation. The first approach does not produce the best algorithm in terms of execution time but it always produce correct answers. The second approach produces better algorithms in terms of execution time and efficiency but it cannot be arbitrarily applied.

The design of efficient algorithms for specific problems and architectures has been already addressed and it continuos being matter of significant research. We think that there must exist a balance between algorithm design and parallel implementation. We showed that the efficient use of virtual processor arrays can be a good midpoint for dynamic programming algorithms.

Problems with variable number of dependencies presents some drawbacks to obtain efficient practical implementation. First, they require a great deal of communication. Second, applying standard block decomposition schemes to that kind of problems leads to unbalanced algorithms where more computational work relies on the processor assigned with the last positions of the DP table.

We suggested to mix communications between processors to obtain only local communications. Also, we have taken advantage of overlapping computation with communication. Finally, by using a non-homogeneous block decomposition approach our experimental results confirm the improvement of the speedup of standard block decomposition techniques for parallel algorithms.

We develop models to predict the performance of homogeneous and non-homogeneous block decomposition schemes. Second order effects, like cache memory and vector units, cause a shift between the observed and the predicted measures. These models need to be further elaborated to accurately predict the performance of very large problems.

There still remain open questions about the mapping of virtual processors to a finite set of physical processors. However, it can be expected that current parallel languages and compiling technology will possitively evolve to provide an effective response to this problem.

Chapter 4

Recurrence Equations

4.1 Introduction

Recurrence equations are important because they are used as the starting specification for parallel algorithms [47] and regular array circuits can be derived by the transformation of recurrence equations [67, 70].

In this chapter we define precisely the concept of recurrence equations. We start with the general case, then we introduce specific cases such as affine recurrence equations, uniform recurrence equations and space-time descriptions. We also define the concept of dependence relations which is of importance for the synthesis of regular array circuits.

4.2 Recurrence Equations

Recurrence equations arefre well known to mathematicians for expressing a large class of functions over a domain D (usually a subset of integers). A recurrence equation specifies $f(n)$ at a point $n \in D$ in terms of f at other points in the domain. That describes a sequence of numbers $a_0, a_1, \ldots, a_n, \ldots$ relating a_n to some of its predecessors. In combinatorial mathematics the primary concern is solving recurrence relations, i.e., addressing the following problem. Given a recurrence equation describing a_n in terms of some other $a's$, determine a 'closed-form' expression for a_n, i.e., an expression for a_n that does not involve any a terms. Our objective here is using the recurrence as an *algorithm* for computing the function and implementing it in a parallel environment.

Let $X, Y, Z, \ldots$ be *identifiers* (also called *variables* or *functions* as we will see later).

A *recurrence equation* defining an identifier X over a domain $\mathcal{D}_i^X$ is an equation of the form

$$X(\mathbf{z}) = g(Y_1(d_1(\mathbf{z})), \ldots, Y_m(d_m(\mathbf{z}))) \quad \forall \mathbf{z} \in \mathcal{D}_i^X \tag{4.1}$$

where

- $\mathcal{D}_i^X$ is the *domain of definition* of the equation. If two or more recurrence equations all have the same identifier X on the left-hand-side of the equation, then their respective domains of definitions are disjoint.

- $d_j, 1 \leq j \leq m$, are *dependency mapping functions* (also called *index mapping functions*) which map $\mathbf{z}$ in $\mathcal{D}_i^X$ to $d_j(\mathbf{z})$

- $Y_j, 1 \leq j \leq m$, are identifiers which can be considered as inputs or can be defined by their own recurrence equations.

- g_i is a strict single-valued function defining the right-hand-side of the equation.

In describing systems of recurrence equations, there are two equally valid points of view which can be taken:

1. A purely functional point of view in which every identifier is a function. A recurrence equation defines a function on the left-hand-side in terms of function in the right-hand-side.
2. Each identifier can be thought of as a single assignment variable and equations equate the variable on the left-hand-side to a function of variables on the right.

4.3 Equation and Identifier Domains

Let $\mathcal{D}_1^X, \mathcal{D}_2^X, \dots, \mathcal{D}_n^X$ be the domains of recurrence equations in whose left-hand-sides the identifier X appears. Then the *domain of identifier* X is $\mathcal{D}^X = \bigcup_i \mathcal{D}_i^X$. As we pointed before, it has to be accomplished that

$$\forall i,j, \quad 1 \le i,j \le n, i \ne j, \quad \mathcal{D}_i^X \cap \mathcal{D}_j^X = \emptyset$$

Domains of definition for recurrence equations of an identifier represent a partition of the domain of the identifier. Since it is typical that all identifiers are defined over the same domain, we will denote it simply as $\mathcal{D}$. That does not necessarily imply that all identifiers are defined over the same partition of the domain.

Usual domains are n-dimensional integer polyhedrons. Whereas a polyhedron is a region containing an infinite number of rational points, a *polyhedral domain* refers to the set of integral points which are inside a polyhedron (or unions of polyhedra). A *polyhedral domain* of dimension n is defined as

$$\mathcal{D} : \{i \in \mathbb{Z}^n, i \in \mathcal{P}\} = \mathbb{Z}^n \cap \mathcal{P} \tag{4.2}$$

where $\mathcal{P}$ is a union of polyhedra of dimension n. A *polyhedron*, $\mathcal{P}$, is a subspace of $\mathbb{Q}^n$ bounded by a finite number of hyperplanes. The so called *implicit definition* of a polyhedron is stated as

$$\mathcal{P} = \{\mathbf{z} | Az = \mathbf{b}, C\mathbf{z} \ge \mathbf{d}\} \tag{4.3}$$

given in terms of equations (rows of A, $\mathbf{b}$) and inequalities (rows of C, $\mathbf{d}$), where A, C are matrices, and $\mathbf{b}$, $\mathbf{d}$, and $\mathbf{z}$ are vectors. $\mathcal{P}$ has an equivalent dual *parametric representation*:

$$\mathcal{P} \quad = \quad \{\mathbf{z} | \mathbf{z} = L\lambda + R\mu + V\nu, \quad \mu, \nu \ge 0, \quad \sum \nu = 1\} \tag{4.4}$$

in terms of a linear combination of lines (columns of matrix L), a convex combination of vertices (columns of matrix V), and a positive combination of extreme rays (columns of matrix R).

Since a specific instance of problem in a finite domain is of no interest at all, we consider here a family of recurrence equations describing a problem with some parameters which can take values in a infinite domain.

4.4 Notation

We use the following compact notation for describing X:

$$X(\mathbf{z}) = \begin{cases} \mathbf{z} \in \mathcal{D}_1^X \rightarrow & g_1(Y_{11}(d_{11}(\mathbf{z})), \dots, Y_{1m_1}(d_{1m_1}(\mathbf{z}))) \\ \mathbf{z} \in \mathcal{D}_2^X \rightarrow & g_2(Y_{21}(d_{21}(\mathbf{z})), \dots, Y_{2m_2}(d_{2m_2}(\mathbf{z}))) \\ \dots & \\ \mathbf{z} \in \mathcal{D}_n^X \rightarrow & g_n(Y_{n1}(d_{n1}(\mathbf{z})), \dots, Y_{nm_n}(d_{nm_n}(\mathbf{z}))) \end{cases} \tag{4.5}$$

It is common to write the conditions $\forall \mathbf{z} \in \mathcal{D}_i^X$, $1 \leq i \leq n$, as predicates, $p_i^X(\mathbf{z})$, which take true or false value (1 or 0) and $\sum_i p_i^X(\mathbf{z}) = 1$, i. e., one and only one of the predicates, p_i^X, takes a true value in a point $\mathbf{z}$. Then, we can rewrite Equation 4.5 as

$$X(\mathbf{z}) = \begin{cases} p_1^X(\mathbf{z}) \rightarrow & g_1(Y_{11}(d_{11}(\mathbf{z})), \dots, Y_{1m_1}(d_{1m_1}(\mathbf{z}))) \\ p_2^X(\mathbf{z}) \rightarrow & g_2(Y_{21}(d_{21}(\mathbf{z})), \dots, Y_{2m_2}(d_{2m_2}(\mathbf{z}))) \\ \dots & \\ p_n^X(\mathbf{z}) \rightarrow & g_n(Y_{n1}(d_{n1}(\mathbf{z})), \dots, Y_{nm_n}(d_{nm_n}(\mathbf{z}))) \end{cases} \tag{4.6}$$

indicating that if $p_i^X(\mathbf{z})$ takes a true value, then, $X(\mathbf{z})$ is evaluated to the corresponding g_i function.

4.5 Dependence Relation

For a system of recurrences, we say that a variable X at a point $\mathbf{u}$ in $\mathcal{D}$ *requires* variable Y at point $\mathbf{v}$, if $\exists i, j$, such that, $d_{ij}(\mathbf{u}) = \mathbf{v}$ and $Y(d_{ij}(\mathbf{u}))$ occurs on the right-hand-side of an equation defining $X(\mathbf{u})$. The transitive closure of this relation is called the *dependence relation* and is denoted by the symbol $\preceq$.

If we ignore variables and we relate points in the domain we can state the dependence relation between points in the domain. We say that a point $\mathbf{u}$ *requires* the point $d(\mathbf{u})$, whenever $d(\mathbf{u})$ occurs on the right-hand-side of an equation of our system. For this reason, functions d_{ij} are called dependency mapping functions. As before, the transitive closure of this relation is called the *dependence relation* and is also denoted by the symbol $\preceq$

4.6 Recurrence Equations Special Cases

A recurrence equation is called an *affine recurrence equation* (ARE) if all dependence functions are of the form $d(\mathbf{z}) = \mathbf{A}\mathbf{z} + \mathbf{b}$, where $\mathbf{A}$ is a constant matrix, and $\mathbf{b}$ is a constant n-vector.

A recurrence equation is called a *uniform recurrence equation* (URE) if all dependence functions are of the form $d(\mathbf{z}) = \mathbf{z} + \delta$, where δ is a constant n-dimensional vector. One can see that uniform recurrence equations are a special case of affine recurrence equations.

A *system of affine recurrence equations* (SARE) is a finite set of mutually recursive affine recurrence equations defining the system variables $X, Y, \dots$ over their respective domains. Variables are designed as either input, output, or local variables of the system. Each variable which is not a system input is defined by a finite set of disjoint recurrence equations in which the variable appears on the left-hand-side. Any variable may appear on the right-hand-side of any equation as often as needed.

In a similar way, a *system of uniform recurrence equations* (SURE) is a finite set of mutually recursive uniform recurrence equations defining the system variables $X, Y, \dots$ over their respective domains with the same considerations as above.

Space-time descriptions are a special case of systems of uniform recurrence equation in which the domain of the system can be written as $\mathcal{D} = T \times \mathbf{S}^{n-1}$, where points $\mathbf{z}$ of the domain can be written as $\mathbf{z} = (t, \mathbf{s}) = (t, s_1, s_2, \dots, s_{n-1})$, t is a 1-dimensional value interpreted as time and $\mathbf{s}$ is a $(n-1)$-dimensional vector interpreted as space. The space describes an $(n-1)$-dimensional array of points which are considered as virtual processors. The uniform dependencies described by the constant vectors $\delta_1, \delta_2, \dots, \delta_d$ are interpreted as communications between virtual processors of the array. They are n-dimensional values which written in the form $\delta = (\Delta t, \Delta s_1, \Delta s_2, \dots, \Delta s_{n-1})$, Δt indicates delays and the $n-1$ remaining values indicate communication to a neighbor processor of $\mathbf{z}$ in the direction $(\Delta s_1, \Delta s_2, \dots, \Delta s_{n-1})$.

4.7 Generating Space-Time Descriptions

Space-time descriptions are used to describe regular array circuits whose implementation is straightforward from that point. To obtain a space-time description from a recurrence equation we have to derive an *allocation function* to map all the index point of the domain to a number of virtual processors and a *timing function* to decide the order in which the index points are going to be computed.

Let us consider a domain defined by an integer polyhedron $\mathcal{P}$ as defined in Equation 4.2. A valid timing function is a function $t : \mathcal{P} \to \mathbb{Z}$ that satisfies the *causality rule*

$$\forall \mathbf{z}_1, \mathbf{z}_2 \in \mathcal{P} : \quad \mathbf{z}_1 \preceq \mathbf{z}_2 \Rightarrow t(\mathbf{z}_1) > t(\mathbf{z}_2) \tag{4.7}$$

i.e, the time at which a computation is done must be greater than the time at which all its dependence points are computed. A valid timing function must have a "beginning time", that is, for all $\mathbf{z} \in \mathcal{P}$, $t(\mathbf{z}) \geq 0$.

Allocation functions determine the set of virtual processors that will perform the computations represented by each point of the polyhedral domain. For every index point of $\mathcal{P}$, $\mathbf{z}$, the virtual processor identified by $a(\mathbf{z})$ will do the computation represented at $\mathbf{z}$. Allocation functions must be conflict free, i.e., two different index points scheduled at the same time have to be mapped to different processors. That is,

$$\forall \mathbf{z}_1, \mathbf{z}_2 \in \mathcal{P} : \quad t(\mathbf{z}_1) = t(\mathbf{z}_2) \quad \Rightarrow \quad a(\mathbf{z}_1) \neq a(\mathbf{z}_2) \tag{4.8}$$

The most common timing and allocation functions are *affine* functions which correspond to a *geometric transformation* of the dependency structure of the original domain. One of the axes in this projected domain is reserved to be the time axis and the remaining ones determine the target (processor) domains.

For uniform recurrence equations is easy to derive affine timing and allocation functions. We can express both functions as a transformation given by the matrix

$$\mathbf{T} = \begin{bmatrix} t \\ \mathbf{A} \end{bmatrix}$$

where

$$t : \mathcal{P} \to \mathbb{Z} \qquad \text{and} \qquad \mathbf{A} : \mathcal{P} \to \mathbb{Z}^{n-1}$$

t represents the time and $\mathbf{A}$ represent the processor array where all the index point of the domain are mapped.

Let $\delta_1, \delta_2, \ldots, \delta_d$ be the set of dependencies on each index point. Every point $\mathbf{z}$ depends on $\mathbf{z} - \delta_i$, for $1 \leq i \leq d$. We can find a linear timing function of the form $t(\mathbf{z}) = \lambda^T \mathbf{z} + \alpha$. In order to find the components of λ we need to solve the constraints obtained by

$$\lambda^T \delta_i > 0 \qquad \text{for} \qquad 1 \leq i \leq d \tag{4.9}$$

Lamport described this method as finding a set of hyperplanes with the same normal direction which partition the entire domain in isotemporal regions; index points scheduled at the same time [51]. The vector λ give us precisely the normal of the hyperplanes.

Once the timing function is determined, possibly several valid allocation functions may exist. We can find linear allocation functions of the form $\mathbf{A}(\mathbf{z}) = \mathbf{S}\mathbf{z}$. It has been shown that valid allocation functions are those which project the domain into a direction different from one orthogonal to the hyperplane normal [68]. Desirable directions of projection are those which project the domain into fewer number of index domains.

When we have an affine recurrence equation we can follow one of two approaches. As Rajopadhye and Fujimuto suggested [70] we still can look for affine timing and allocation functions solving the restrictions obtained by applying the causality property to affine dependencies. Then, we apply a pipelining technique to obtain a regular array. This approach does not always succeed. In their paper, these authors show an example with an affine recurrence equation with no affine timing function at all. The other approach deals with restating the recurrence equation in such a way that uniform dependencies can be obtained. This process called *uniformization* is more general. However, at present there is no general procedure make arbitrary recurrence equations uniform.

4.8 Conclusions

We have shown a general definition for recurrence equations over index domains. We have distinguished between domain of definition and domain of an identifier. An identifier can be defined by several recurrence equations. The union of domains of definition represent the domain of a variable. Common domains are restricted to n-dimensional integer polyhedrons. Since special cases of recurrence equations depend on the shape of dependency mapping functions, we introduced the dependence relation between variables and between points in the domain. Special cases, affine, uniform and space-time recurrence equations, were defined. Finally, we have shown how to obtain space-time descriptions from uniform recurrence equations. In chapter 6 we will discuss the process of making recurrence equations uniform.

Chapter 5

Dataflow Analysis of Parallel Expressions in C*

5.1 Introduction

Dataflow analysis is more or less the estimation of the effects caused by program statements. This estimation is based on two things: an abstraction of the information needed as pre-requisite for the optimization transformation, and the propagation of the information along the statements of the source program. Generally speaking, dataflow analysis determines if one value written (defined) to a variable in some part of a program is read (used) in another part of the program.

Dataflow information can be used to convert imperative programs to single assignment code. In this chapter we focus on applying dataflow analysis to parallel statements made in C* in order to obtain the recurrence equations that describe the computations performed by some nested loops of parallel statements. It has been shown that recurrence equations are an essential intermediate step to describe algorithms suitable for hardware implementation [16]. The variety of loops that can be expressed with parallel statements is wide, so we restrict here ourselves to the class of loops used to program dynamic programming algorithms in C*. We have focused on the C* programming language for two reasons. First, it allows us to express that kind of computation in a clean and compact way. Second, its syntax allows us to restrict the analysis to specific parts of the code, avoiding spending time in other parts.

The general process of obtaining SREs from data-parallel code requires of a thoroughgoing dataflow analysis. The first step of the technique is determining the flow dependencies between parallel data in C* programs. Therefore, we need to port the dataflow analysis technique for imperative sequential code to parallel loops consisting of data-parallel statements. Since we are dealing now with parallel code we need to make some observations about the sequencing of instructions, and we have to restate some definitions commonly used in sequential programming. The second step uses the flow dependency information for building recurrence equations describing the functional behavior of the program.

Therefore, we present here a dataflow analysis technique to obtain SREs from loops consisting of data-parallel statements. The wide variety of loops that can be generated following such a general statement forces us to restrict our analysis to the class of loops obtained implementing dynamic programming solutions of some problems. That class of problems is of importance because problems belonging to it have been successfully implemented as systolic algorithms and the process to do that has been widely studied.

In section 5.2 we present all the machinery needed for our data flow analysis technique. In section 5.3 we show some results of it. Finally, some conclusions are made at the end of this chapter.

```
shape []...[] Shape;    /* Shape definition */
...:Shape var;          /* parallel variable declaration */
with( Shape ) {         /* Current shape operation */

  ... /* initialization */

  for( k = 1; k <= N; k++ ) {
    where( .... ) {     /* Active positions selection */
    ...
    ...                 /* References and assignments to parallel vars. */
    }
  }
}
```

Figure 5.1: General shape of code segments being analyzed in the dataflow analysis algorithm.

5.2 Definitions and Notation

Let us make some definitions of terms we need to explain both steps of the algorithm to obtain recurrence equations from parallel code which will be explained in the next section. First, let us start explaining the general structure of the loops that we will consider which is shown in Figure 5.1.

We declare a parallel variable of a given shape. The use of the parallel variable is delimited by the statement **with**. The body of this code segment consists of an initialization block typically done by every data of the parallel variable. The next part consists of a sequential loop in whose body reference to the parallel data is made. Sometimes we restrict the activation of parallel data by introducing the statement **where** which indicates a condition. All positions satisfying this condition perform the code delimited by the statement.

In the following we define precisely the code fragments that we will analyze, the iteration vector for parallel variables, the flow dependency between parallel variable statements, the source function of a data reference, the sequencing predicate in parallel code and the dependency relation in parallel variable statements.

5.2.1 Affine C* Program Fragment

The kind of loops that we can describe with the general shape above is broad and sometimes complex to analyze. Therefore, we need to make some restrictions on program fragments in order for they to be analyzable. Fortunately, even with this restriction the scope of programs we can describe is broad and of relevance for scientific applications. The definition we are going to give is similar to that presented in [29] with some arrangements to parallel code.

An affine C* program fragment (ACPF) is a code segment enclosed within a **with** statement consisting of a initialization part followed by a loop nest such that in every statement explicit or implicit subscript functions, conditions in **where** statements, and loop bounds are affine functions of loop variables and symbolic constants.

A recurrence equation per parallel variable which is defined and used within the an ACPF will be constructed. Variables used, but not written, within the ACPF will be considered as symbolic constants.

5.2.2 Notation

Before we proceed to describe the problem and how to solve it, it is necessary to introduce some notation.

A *vector* (also called a *tuple*) is simply an ordered set of integers. Vectors are denoted with bold letters, such as $\mathbf{w}$, $\mathbf{r}$, $\mathbf{s}$. They are used to represent points in an n-dimensional space. The smallest unit of computation that we consider in this chapter is a statement instance $W[\mathbf{w}, \mathbf{s}]$ that is specified by W -statement of the program, $\mathbf{w}$ - vector of loop variables values (loops which surround the statement W are included), and by $\mathbf{s}$ -vector of symbolic constants. We call a variable a *symbolic constant* if it is not a loop variable and it is not assigned in the fragment of the program we analyze.

We say that vector $\mathbf{w}$ is lexicographically less than vector $\mathbf{r}$, denoted by $\mathbf{w} \ll \mathbf{r}$, iff $\exists j$ such that $w_j = r_j$, for $1 \leq j < i$, and $w_i < r_j$.

A *relation* is a set of ordered pairs of vectors. $(\mathbf{w} \rightarrow \mathbf{r}) \in R$ means that pair $(\mathbf{w}, \mathbf{r})$ belongs to the relation R.

5.2.3 Domain of Parallel Variables

In an ACPF we are interested in parallel variables being defined. The first thing we need to consider is that parallel variables hold many values, one per entry in the shape declaration. In each point, a parallel variable can itself be an array of values or scalar. Finally, given the imperative nature of C* programs, parallel variables can be multiply assigned within an ACPF. Typically, when they appear in loops, they are assigned in every iteration, possibly holding different values at different iterations. Informally, the domain of a parallel variable is the space in which the variable can take values.

Let us consider all statements of the form

```
S: P = . . . Q . . .
```

which appear inside a nest loop body. To determine the domain in which variable P is defined we will consider

- an index per nest loop embedding,
- an index per shape dimension of P, and
- as many indexes as array parallel variables were declared

in the recurrence equation for P. Then, a point, $\mathbf{z}$ of the domain for P, is described as

$$\mathbf{z} \quad = \quad (\mathcal{I}, \mathbf{s}, \mathbf{a}) = \{i_{l_1}, \ldots, i_{l_n}, i_{s_1}, \ldots, i_{s_m}, i_{a_1}, \ldots, i_{a_p}\}$$

where $i_{l's}, i_{s's}$ and $i_{a's}$ refer to loop nest, shape indexes and array dimensions, respectively. Vector $\mathcal{I}$ is the iteration vector of the statement recognized as the dynamic domain of P, and $(\mathbf{s}, \mathbf{a})$ is considered as the static (declared) domain of P.

5.2.4 Source Function

In general, for any reference to a parallel variable, we have to determine which statement produced the value that is being read.

For a given statement instance $S_2[\mathbf{r}]$ the *source function* produces the coordinates, iteration vector, of the statement instance $S_1[\mathbf{w}]$ such that $S_1[\mathbf{w}]$ supplies the value used in $S_2[\mathbf{r}]$.

The source function can be represented in several ways. For example, Feautrier used *quasi-affine selection tree* [29]. Maydan *et al* defined *last write tree* to represent the same information [56]. Mazlov [58] and Pugh [65] separately defined *dependence relations* as another way to represent this information. Given that dependence relations are close to the notation used in recurrence equation, we will used them to describe dataflow information.

5.2.5 Flow Dependency

Let us consider the following program fragment

```
A: P = . . .
   . . .
B: Q = . . . P . . .
```

we say there exists a *flow dependency* from a parallel variable access $A(\mathcal{I})$ to a parallel variable access $B(\mathcal{I}')$ if

- A is executed with iteration vector $\mathcal{I}$,
- B is executed with iteration vector $\mathcal{I}'$,
- $A(\mathcal{I})$ writes to the same location as is read by $B(\mathcal{I}')$,
- $A(\mathcal{I})$ is executed before $B(\mathcal{I}')$, and
- There is no write to the location read by $B(\mathcal{I}')$ between the execution of $A(\mathcal{I})$ and $B(\mathcal{I}')$.

5.2.6 Sequencing Predicate

We say that instance of statement W specified by loop variables vector $\mathbf{w}$ and symbolic constants vector s is *executed before* instance of statement R specified by loop variables vector $\mathbf{r}$ and symbolic constants vector s, denoted by $W[\mathbf{w},\mathrm{s}] \ll R[\mathbf{r},\mathrm{s}]$, iff

$$\mathbf{w}[1..n] \ll \mathbf{r}[1..n] \vee (\mathbf{w}[1..n] = \mathbf{r}[1..n] \wedge W \ll R)$$

5.2.7 Dataflow Dependency Relation

The dataflow dependence relation *dfRel* that describes the dependencies coming to the read reference $R.A$ of statement R is defined by the following:

$$\begin{aligned}
&\forall \mathbf{r},\mathrm{s} : (V[\mathrm{v},\mathrm{s}] \rightarrow R.A[\mathbf{r},\mathrm{s}]) \in dfRel(\mathbf{w},\mathbf{r},\mathrm{s}) \Leftrightarrow \\
&\quad V[\mathrm{v},\mathrm{s}] = \max\nolimits_{\ll}(\quad W[\mathbf{w},\mathrm{s}] | \mathbf{w} \in [W,\mathrm{s}] \wedge Arr(W.B) = Arr(R.A) \wedge \\
&\qquad\qquad\qquad\qquad W.\mathbf{B}(\mathbf{w},\mathrm{s}) = R.\mathbf{A}(\mathbf{w},\mathrm{s}) \wedge W[\mathbf{w},\mathrm{s}] \ll R[\mathbf{r},\mathrm{s}])
\end{aligned}$$

Since this definition is constructive, we can use it to compute the dataflow relation. When the lexicographical maximum is computed, the result is a dataflow relation which is represented as a union of the following m simple dataflow relations:

$$dfRel = \{W_i[\mathbf{w},\mathrm{s}] \rightarrow R.A[\mathbf{r},\mathrm{s}] | \quad \text{for some} \quad i \quad \text{and any} \quad \mathbf{w},\mathbf{r},\mathrm{s} \quad \text{such that} \quad dfRel_i(\mathbf{w},\mathbf{r},\mathrm{s})\} \tag{5.1}$$

where each $dfRel_i$ is a conjunction of constraints and

$$\bigcup_{i=1}^{m} \pi_{\mathbf{r},\mathrm{s}}(dfRel_i(\mathbf{w},\mathbf{r},\mathrm{s})) \subseteq [R,\mathrm{s}] \tag{5.2}$$

Since source functions may involve integer division by a constant and we want to keep conjuncts $dfRel_i$ affine, we use wild-card variables to represent the integer division. That is, we replace the constraint $i = \lfloor k/c \rfloor$ with the affine constraint $(ci + \alpha = k) \wedge (0 \leq \alpha \leq c-1)$.

5.3 Deriving Recurrence Equations

In this section we will show how to obtain recurrence equations from C* code implementing solutions to dynamic programming problems. We begin by showing that all such C* programs have the following basic structure:

```
shape []...[] Shape;   /* Shape definition */
...:Shape var;     /* parallel variable declaration */
with( Shape ) {        /* Current shape operation */

  ... /* initialization */

  for( k = 0; k < N; k++ ) {
    where( .... ) {    /* Active positions selection */
    ...
    ...                /* References and assignments to parallel vars. */
    }
  }
}
```

The research done by Feautrier [29] describes a dataflow analysis algorithm for array and scalar references in imperative code consisting of sequential loops with static control. It can be extended to include parallel variables like C* introduces. The advantage of using data-parallel code instead of sequential imperative code is that the former restricts the loop's depths for the dataflow analysis. Furthermore, because C* operates on parallel data from only one shape at a time, the dataflow analysis can be restricted to the scope of the corresponding **with** statement. Feautrier's algorithm can be briefly summarized as follows. For a given reference to an array, scalar or parallel variable **M** in a statement s, construct the candidate list from all pairs $\langle r, p\rangle$ where r is a statement which modifies **M** and p is the dependence depth. Order the candidate list by decreasing depth. For each candidate, test if there is a possibility that it will contribute to the final source function. If not, discard the candidate. Otherwise, compute the direct dependence by applying a linear programming algorithm known as Parameteric Integer Programming. Feautrier describes a technique to combine direct dependencies to obtain recurrence equations. The details of the algorithm can be found in [29].

To obtain recurrence equations from C* we use the following rules:

1. A recurrence equation per variable which is both defined and used within a loop will be considered.

2. Variables read, but not written, within loops will be considered as constants.

3. An index per shape dimension of a parallel variable will be considered in the recurrence equation.

4. Array variables will add to the recurrence equation as many indices as were declared.

5. An index per nested loop embedding the definition of a variable will be considered in the recurrence equation.

6. Constraints for recurrence equations are determined by the bounds in the nested loops and the bounds on the shape declaration.

7. Conjunctive constrains will be added every time a **where** statement is found.

5.3.1 Shortest-Path Problem

In Chapter 2 we introduce the shortest-path problem. Here we present an ACPF to solve the problem. We need to recall that the solution to this problem can be computed from a sequence of matrix-vector multiplications. A well known data-parallel algorithm for matrix-vector product distributes each row of the matrix and the corresponding element of the vector to each processor. m processors are required in this case. Each processor computes a single element of the resulting vector, but it requires all the elements of the input vector. So, each processor initially computes the term associated with the vector element it contains and then this vector element is moved to the previous processor. Processors are connected in a ring, so the first processor moves its element to the last processor. After m steps, each vector element has visited every processor so the final matrix-vector product has been computed. Given that the result is distributed among m processors, the next matrix-vector product can be initiated after updating the matrix elements in each processor.

A piece of a C* program to compute this sequence of matrix-vector products is:

```
shape [N] Rows;
struct cell_data {
    int r[N];
    int a;
    int b;
  };

struct cell_data:Rows P;

with( Rows ) {
  for( k = 0; k < N-1; k++ ) {
    /* Mk matrix initialization */
    P.r = ...
    . . .

    /* Ck computation */
    P.a = 0;
    for( i = 0; i < M; i++ ) {
      /* Single term computation */
      P.a += (P.r[(i+pcoord(0)) %% N]*P.b);

      /* Vector element communication */
      P.b = [(.+1) %% N]P.b;
    }

    P.b = P.a;
  }
}
```

What information can a compiler extract from this program ? First, note that references to parallel variables are explicitly made by the construct `with`. It consists of a loop whose body updates and uses the values of parallel variables. So every time a parallel variable is found, an index in a recurrence equation should be introduced.

Three main variables are used in the program. There is an array per processor, `P.r`, defined within the outmost nested loop and is used inside the most nested loop. It is used to stored a single row of the corresponding stage's matrix. Given that it is only used within the second loop and its reference is based on a affine transformation of the second loop's index, there exists an alignment between this array and the second loop's index. As we will show later, this alignment can be explicitly defined in the array assignment in order to simplify the recurrence equations.

The variable `P.a` is used to compute a dot product between a single row and the input vector. It is initialized outside the second loop and it is redefined inside the second loop. The variable `P.a` is originally defined outside the outmost nested loop and it is redefined inside the inmost nested loop. Its redefinition is based in a communication to the next processor, assuming that the processors are ring interconnected. Applying data-flow analysis to these variables we can obtain the following recurrence equations where the indexes represent the nested loop's indexes and the parallel variable index:

$$P.r(k,i,p) = \begin{cases} i = 0 \rightarrow & ct \\ 0 < i \leq M \rightarrow & P.r(k,i-1,p) \end{cases} \tag{5.3}$$

$$P.a(k,i,p) = \begin{cases} i = 0 \rightarrow 0 \\ \\ 0 < i + p - 1 \leq M \rightarrow \\ \quad P.a(k, i-1, p) + P.r(k, i+p, p) * P.b(k, i-1, p) \\ \\ i + p - 1 > M \rightarrow \\ \quad P.a(k, i-1, p) + P.r(k, i+p-M, p) * P.b(k, i-1, p) \end{cases} \tag{5.4}$$

$$P.b(k,i,p) = \begin{cases} i = 1 \rightarrow \\ \begin{cases} k = 0 \rightarrow & c \\ 0 < k < M \rightarrow & P.b(k-1, M+1, p) \end{cases} \\ \\ 1 < i \leq M \rightarrow \\ \begin{cases} 0 < p+1 < M \rightarrow & P.b(k, i-1, p-1) \\ p + 1 = M \rightarrow & P.b(k, i-1, 0) \end{cases} \\ \\ i = M + 1 \rightarrow P.a(k, M, p) \end{cases} \tag{5.5}$$

where c means a constant value. It is easy to see the inherently sequential nature of the k loop, so nothing can be done to extract parallelism at this level. So, now we have to consider only the recurrence equations defined over the $i - p$ plane. Note that $P.r$ does not change during i's loop. To simplify the equation we can consider it as a constant. Our recurrence equation can now be written as:

$$P.a(i,p) = \begin{cases} i = 0 \rightarrow 0 \\ \\ 1 < i \leq M \rightarrow \\ \quad P.a(i-1, p) + P.r_c * P.b(i-1, p) \end{cases} \tag{5.6}$$

$$P.b(i,p) = \begin{cases} i = 1 \rightarrow ct \\ \\ 1 < i \leq M \rightarrow \\ \begin{cases} 0 < p+1 < M \rightarrow & P.b(i-1, p-1) \\ p + 1 = M \rightarrow & P.b(i-1, 0) \end{cases} \\ \\ i = M + 1 \rightarrow P.a(i-1, p) \end{cases} \tag{5.7}$$

Except for the dependency (i-1,0), all dependencies are now local. If we assume that the array has wraparound connections, we can think of this dependency as uniform.

5.3.2 Longest Common Subsequence

Let us now consider the longest common subsequence problem introduced in Chapter 2. Let us try to see what a data-parallel program looks like for this example.

Although there are more efficient ways to implement the construction of the F table [49], the following code is a clear way to express such that process

```
shape [N+1][M+1] Table;
shape [] String;

int:Table F, ii, jj;
```

```
char:String s1, s2;

with( Table ) {
  where( pcoord(0) == 0 ) F = 0;
  where( pcoord(1) == 0 ) F = 0;

  for( k = 0; k < M+N; k++ ) {
    where( ((pcoord(0) + pcoord(1)) == k) &&
   (pcoord(0) > 0) && (pcoord(1) > 0) ) {
      ii = pcoord(0) - 1;
      jj = pcoord(1) - 1;
      where( [ii]s1 == [jj]s2 )
        F = [.-1][.-1]F + 1;
      else
        F = ([.-1][.]F >? [.][.-1]F);
    }
  }
}
```

Note that a two-dimensional array of virtual processors is used. A processor is used for each table entry. Initially, all processors along the first row and the first column are set to zero. Then, all the entries are filled. The first **where** statement restricts the active processors in each step. It selects those processors along the diagonal $i + j = q$, excluding the processors in the first row and the first column. In Figure 4a. we can view the the wavefront scheduling policy used for filling the table.

Using our dataflow analysis technique to obtain recurrence equations we have to use one index per processor dimension and one index per loop. Given that F is a scalar variable per processor, no index is required for it.

$$F[k,p,q] = \begin{cases} p = 0 \rightarrow 0 \\ q = 0 \rightarrow 0 \\ 1 \leq k \leq M + N \wedge 1 \leq p \leq M \wedge 1 \leq q \leq N \rightarrow \\ \begin{cases} p + q = k \wedge s1[p] = s2[q] \rightarrow & F[k-2, p-1, q-1] + 1 \\ i + j = k \wedge s1[p] \neq s2[q] \rightarrow & \max\{F[k-1, p-1, q], F[k-1, p, q-1]\} \end{cases} \end{cases} \tag{5.8}$$

This is a uniform recurrence equation, but it is defined over three dimensions. From the program it can be seen that the k-direction represents a schedule. Given that the domain is a plane, at any single value of k there is a line lying in the plane, $p + q = k$. So, we can project this domain onto the $k - q$ plane to obtain the following recurrence equation.

$$F[k,q] = \begin{cases} i = 0 \rightarrow 0 \\ j = 0 \rightarrow 0 \\ 1 \leq k \leq M + N \wedge \max\{1, N-k\} \leq q \leq \min\{k, M\} \rightarrow \\ \begin{cases} s1[k-q] = s2[q] \rightarrow & F[k-2, q-1] + 1 \\ s1[k-q] \neq s2[q] \rightarrow & \max\{F[k-1, q-1], F[k-1, q]\} \end{cases} \end{cases} \tag{5.9}$$

Eq. (5.9) is itself a uniform recurrence equation where k can be interpreted as time and q can be interpreted as space.

5.3.3 All-Pairs Shortest Path

Putting the recurrence equation of the all-pairs shortest path problem (seen in Chapter 2) in a data-parallel program using n^2 virtual processors is straightforward. The following C* code computes such a recurrence equation.

```
shape [N+1][N+1] Mesh;
int:Mesh M;

with( Mesh )
  for( k = 1; k <= N; k++ )
      M = (M <? ([.][k]M + [k][.]M);
```

The recurrence equation obtained from this code is

$$M(k,p,q) = \begin{cases} k = 0 \rightarrow ct \\ 1 \leq k \leq N \rightarrow \\ \quad \min\{M(k-1,p,q), M(k-1,p,k) + M(k-1,k,q)\} \end{cases} \tag{5.10}$$

which is just the single assignment version of the algorithm presented in [50].

5.3.4 Optimal Matrix Parenthesization

Efficient mappings of the systolic algorithm to 2D meshes can be found in [46] for this problem. However, for clarity we will start with a straightforward implementation in a C* program of the recurrence equation of the OMP problem shown in Chapter 2.

```
shape [N][N] Mesh;
shape [N+1] Dims;

int:Mesh m, ii, jj, kk;
int:Dims r;

with( Mesh ) {
  where( pcoord(0) = pcoord(1) )
    M = 0;

  for( d = 1; d <= N; k++ ) {
    where( (pcoord(1) - pcoord(0)) == d ) {
      ii = pcoord(0); jj = pcoord(1) + 1; kk = pcoord(0) + 1;
      M = [.][pcoord(0)]M + [.+1][.]M + [ii]r*[kk]r*[jj]r;
      for( k = 1; k < d; k++ ) {
        kk = pcoord(0)+k+1;
        M <? = [.][pcoord(0)+k]M + [.+k+1][.]M + [ii]r*[kk]r*[jj]r;
      }
    }
  }

}
```

Several criticisms can be made of the previous program. First, note that although a two-dimensional array is declared, only $n(n+1)/2$ processors are really used; one processor per entry in the triangular table. They are used only across diagonals. That means that those processors for which $j - i = d$ are active only once during the outermost loop. During subsequent iterations of the outermost loop, fewer processors are active, and it takes more time to compute their results in the innermost loop. Finally, because of restrictions on the C* language, we have to use three parallel variables (`ii, jj, kk`) to parallel index the array of dimensions `r`.

In obtaining the system of recurrence equations from this C* code we require four indices for the expressions: two for array dimensions, and one per nested loop. Because `M` is a scalar value per processor, no additional index is required. Although a simple dataflow analysis would generate a recurrence equation per ii, kk, jj, and M variables, a more sophisticated analysis would simplify the equations eliminating those for ii, kk, and jj variables. Such an analyisis would obtain:

$$M(d,k,p,q) = \begin{cases} p = q \rightarrow 0 \\ 1 \leq d < N \wedge q - p = d \\ \quad \begin{cases} k = 0 \rightarrow M(0,k,p,p) + M(d-1,d-1,p+1,q) + \\ r(p) * r(p+1) * r(q) \\ 1 \leq k < d \rightarrow \min\{M(d,k-1,p,q), \\ M(k,k,p,p+k) + M(d-k-1,d-k+1,p+k+1,q) + \\ r(p) * r(p+k+1) * r(q)\} \\ k = d \rightarrow M(d,k-1,p,q) \end{cases} \end{cases} \tag{5.11}$$

A careful examination of the previous equation can show us that although it is defined over a 4-dimensional space, all points satisfy $q-p=d$, $0 \leq d < N$, $0 \leq k < q-p$. Projecting along $(1000)^T$ direction and reindexing we can obtain the following recurrence equation:

$$\hat{M}(k,p,q) = \begin{cases} p = q \rightarrow 0 \\ 1 \leq q - p < N \\ \quad \begin{cases} k = p \rightarrow \hat{M}(p,p,p) + \hat{M}(q,p+1,q) + \\ r(p) * r(p+1) * r(q) \\ p < k < d \rightarrow \min\{\hat{M}(k-1,p,q), \\ \hat{M}(k,p,k) + \hat{M}(q,k+1,q) + \\ r(p) * r(k) * r(q)\} \\ k = d \rightarrow \hat{M}(k-1,p,q) \end{cases} \end{cases} \tag{5.12}$$

Here, the scheduling along diagonals is ignored and the serialization of the min computation is done by varying k exactly from p to $q-1$.

5.4 Related Work

Dataflow analysis of scalar variables has been performed by compilers to optimize the code generated by them [3]. When parallel processing emerged, many techniques were proposed to automatic parallelize sequential programs. Dependence analysis was proposed as a way to determine if different instances of loops could be executed in parallel [10]. Since dependence analysis only determines if two different references access the same memory location, it results were somewhat limited [57]. Recently, dataflow analysis of array references has been proposed as a technique to determine if sequential loops can be parallelized using array privatization. Feautrier [29] developed a method to analyze sequential affine program segments with static control. He showed that his method, based on linear integer programming, can be used for array privatization, for converting imperative sequential code to single assignment code, and to program checking and parallel program construction.

Maydan *el al* [56] developed a new algorithm to obtain the same information in simplified cases. Based upon known benchmarks, they demonstrated that a high percentage of dataflow array references can be expressed or derived from simplified cases.

Recently, Pugh and Wannacot [65] developed an algorithm to perform the same optimizations based on the techniques that the first author developed for dependency analysis [64]. Their method simplifies expressions, Pressburguer formulas, that model value based dependencies. Mazlov [58] developed a method whose main characteristic is the order in which a given reference can determine the source function, i.e., the statement producing the value read by the reference. Although, previous methods had similar objectives, the kind of information they used to achieve them, was different. Feautrier introduced some expressions he called *quasts*, *Quasi-affine selection trees*. Maydan designed a data structure to hold the same information, *last write trees*. Pugh and Mazlov used *dependency relations* to model the same information.

5.5 Conclusions

A dataflow analysis technique for parallel expression in C* has been presented. It is based on the concept of dataflow dependence relation. This technique is used to obtain the recurrence equations that model the behavior of affine C* program fragments. Besides these restrictions, the technique is general enough to be applied to a broad class of problems. Program fragments for dynamic programming algorithms are simple and can be of one of two types: serial and non-serial data-parallel loops. The application of the dataflow analysis technique to dynamic programming algorithms is effective. Recurrence equations are obtained converting C* program fragments into single assignment code.

Chapter 6

The Uniformization of Recurrence Equations

Systems of recurrence equations have been used as high level descriptions of computations suitable for systolic processing. The first step in systolic synthesis is to transform a system of recurrence equations into one where all the dependencies have constant displacements. This process cannot be achieved in the general case. Furthermore, even when such a transformation is possible, it usually involves some clever steps requiring human expertise. In chapter 4 we introduced the concept of recurrence equation and its special cases. In this chapter we examine the uniformization process for recurrence equation through some examples. We propose the use of a set of mathematical transformations working over polyhedral domains combined with visualization facilities to explore and derive the right set of transformations.

6.1 Introduction

Recurrence equations describe computations to be performed on index points belonging to a certain domain. In the process of systolic synthesis one of the important steps is to transform a recurrence equation into a shape in which all its dependencies are uniform [67].

The uniformization process cannot be achieved for arbitrary recurrence equations. Rajopadhye and Fujimoto showed that recurrence equations with affine dependencies, i.e. where each $d(\mathbf{z})$ is of the form $A\mathbf{z} + \mathbf{b}$, A and $\mathbf{b}$ constants, can be effectively transformed to a uniform shape [70]. However, even when this process is feasible, the set of transformations usually requires clever steps requiring human intervention.

The uniformization process involves alignment of variables to a suitable reference point, localization of long communications between index points in the recurrence equation, and serialization of long computation to allow a single index point to perform a fixed amount of computations depending only on a fixed number of inputs. Generally, those steps are done by transforming the original domain into a new one preserving the dependencies among index points, and by manipulating the computations performed in the index points. Many types of transformations can be used to achieve that goal. Important ones include transformations over polyhedral domains such as domain shifting, skewing, splitting, and projecting, and transformations to manipulate abstract syntax trees, such as adding new variables and pipelining values along certain directions.

To derive the right set of transformations, the designers frequently draw graphical representations of the problem's dependency graph. Several tools have been proposed to aid in the process of systolic design [19, 52]. Some of them have shown that as soon as a system of uniform recurrence equations can be obtained, systematic transformations can be applied to obtain different systolic implementations [60]. Crystal [19] and ALPHA [52] are approaches based on functional programming. Both provide the designer an equational language to specify recurrence equations and a set of transformations to transform an input program into an equivalent output one. However, in both cases little help is provided to the user to determine the right transformation to be applied at any point. That makes them suitable only for experienced users familiar with the problem.

A third approach followed by Ribas [72] and Barnett [11] is based on the derivation of systolic programs from input programs composed, generally, of nested loops. If the input program contains only uniform dependencies, Ribas' process is automatic. On the other hand, Barnett assumes that a program and a valid systolic schedule are given and his compiler generates systolic programs for multicomputers. Currently there are no general purpose tools to perform the job of uniformizing a system of recurrence equations in a completely automatic way, and the existing ones provide little help to decide the right transformation to apply at each step.

As can be inferred, once we get uniform recurrence equations we can apply automatic methods, such as Ribas' and Barnett's approaches, to derive systolic algorithms. Therefore the main goal is to obtain uniform recurrence equations from arbitrary ones.

In this chapter we present some examples of the uniformization process of recurrence equations. We highlight the key steps in doing such a process to exhibit the difficulty of doing the general process in automatic ways. We propose the use of an adequate set of mathematical transformations combined with some visualization facilities that allow exploring different transformations before we derive the right set of transformations.

We briefly discuss a set of useful transformations for the uniformization process. Then, we examine the uniformization process for the transitive closure problem and we generate a space-time representation for it. The necessity for visualization facilities will be evident as well as the experience of the user in performing such a process. Finally, we present the uniformization process for the gap problem which, to the best of our knowledge, no previous work on such a problem has been reported.

6.2 A Transformation as The Basic Block

In chapter 4 we introduced recurrence equations. From Eq. 4.1, we can see that a recurrence equation can be described by variables, domains and computations. Variables represent values which are associated with each point within its domain. Domains are usually restricted to unions of polyhedra. Computations represent the operations performed at each point of the domain with its associated variables.

Our primary concern is to determine (when it is possible) a sequence of transformations, T_1, T_2, ..., T_n such that when applied to a recurrence equation RE, we obtain a sequence of new recurrence equations, $RE_1, RE_2, \ldots, RE_n$, trying to give RE_n a uniform shape.

$$RE \xrightarrow{T_1} RE_1 \xrightarrow{T_2} RE_2 \xrightarrow{T_3} \cdots \xrightarrow{T_{n-1}} RE_{n-1} \xrightarrow{T_n} RE_n = URE$$

Transformations between recurrence equations can manipulate variables and/or computations. That implies that useful transformations must include mappings to manipulate variables, abstract syntax trees and polyhedral domains. In general, we can consider a transformation as a mapping

$$T : \{(\mathcal{F}, \mathcal{V}(\mathcal{D}))\} \rightarrow \{(\mathcal{F}, \mathcal{V}(\mathcal{D}))\}$$

where $\mathcal{F}$ stands for computations and $\mathcal{V}(\mathcal{D})$ stands for variables defined over domain $\mathcal{D}$.

Before we describe some efficient ways to represent variables, domains and computation, we present the most frequently used transformations to manipulate recurrence equations.

6.3 A Set of Useful Transformations

The set of transformations are referred to variable, domain, and computation manipulation.

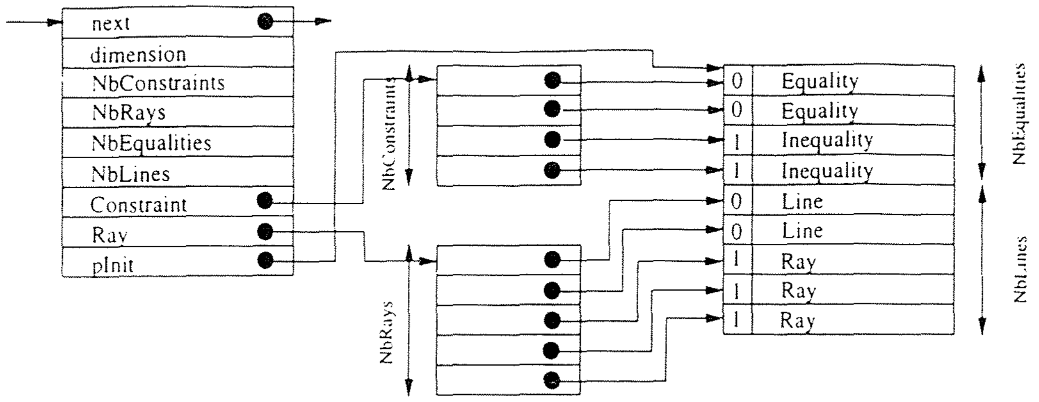


Figure 6.1: The data structure used to represent unions of polyhedra.

6.3.1 Variable Manipulation

Usually we need the following transformation to handle variables:

- *New Variables.* We must introduce new variables over some domains.
- *Alignment.* It is common to place original variables according some points of the domain; usually, points located near the domain's boundary.

6.3.2 Domain manipulation

A useful set of domain transformations would require a useful set of manipulating operations over polyhedra. Since implicit and parametric representation are useful for different transformations, a suitable data structure is required to represent unions of polyhedra; in Figure 6.1 we show the data structure used for that. Both contraints and rays are included, which makes the representation redundant. However, they are used to perform different operations. There are some functions that allow us to transform between representations.

Wilde [80] reports a library of useful operations for manipulating polyhedra. Three kinds of operations are useful:

- *Transformation between representations.* A kind of operation particularly important is that of dealing between different ways of polyhedral representation. For example, we can describe a polyhedral domain in terms of linear equalities and inequalities, and later on we might wish to obtain a description of the same domain based on geometric features of the polyhedron (lines, rays, vertices). The convex hull problem is a well known problem dealing with computing the facets of the convex hull surrounding a given set of points.
- *Operation between domains.* Typical operations to manipulate sets; unions, intersections and differences are required to compute more complex operations.
- *Projections.* Transformations applied to change a given domain $\mathcal{D}$ into another domain $\mathcal{D}'$. Image and preimage of affine mapping functions, $Tx + t$ are the basic projections to build more complicated projections like domain shifting and skewing.

In Figure 6.2 we show the most frequent domain transformations, image and pre-image computation. The way to do it is based on the different types of representing a polyhedron.

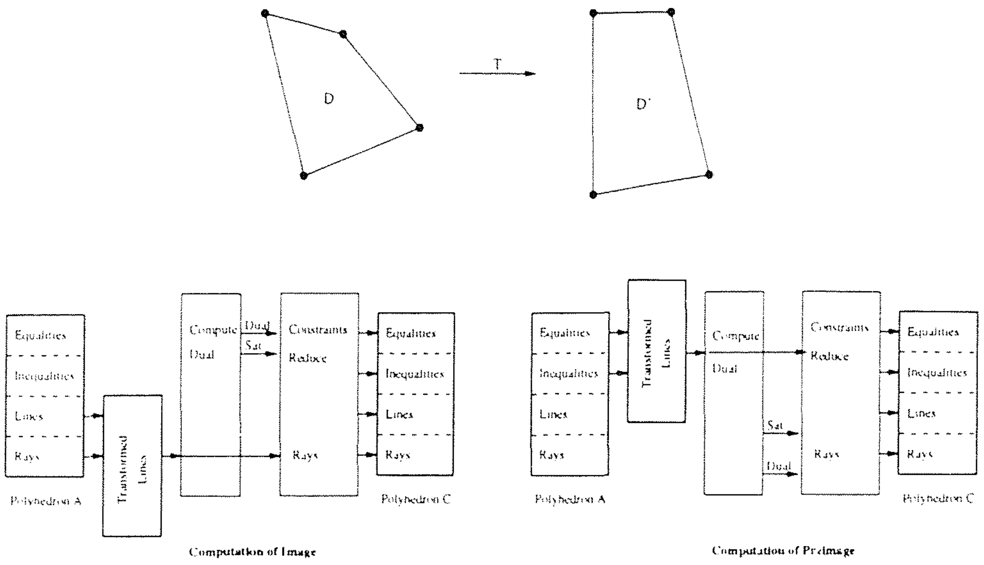


Figure 6.2: A linear transformation, image, of domain D into D' uses the parametric representation. The inverse transformation, pre-image, is computed from the implicit representation.

$$A = \sum_{i=1}^{N} x_i$$

$$\begin{cases} X[i] = \begin{cases} i = 0 \rightarrow & 0 \\ 1 \leq i \leq N \rightarrow & X[i-1] + x[i] \end{cases} \\ A = X[N] \end{cases}$$

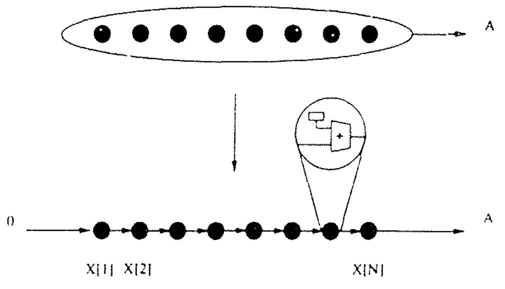


Figure 6.3: Serialization of an expression with a non-fixed number or parameters.

6.3.3 Computation manipulation

Two kinds of manipulations are useful: pipelining and serialization. Both seem to be the same, but their difference appears in the context where they are applied.

- *Serialization.* It is used to distribute a variable size computation performed in a single index point along different points lying in a certain direction in such a way that each single point computes a fixed computation from a fixed number of arguments (inputs). This operation involves a communication pattern among all the points lying in the specified direction. A typical serialization operation is showed in Figure 6.3.

- *Pipelining.* It is used to communicate values required in a certain index point produced by index points separated by non-fixed distances. Normally, it is done by nearest neighbor communications along index points specified in a certain direction. At difference with serialization, intermediate index points do not perform any computation on pipelined values as can be observed in Figure 6.4.

$$A = \cdots x \cdots$$

$$\begin{cases} X[i] = \begin{cases} i = 1 \rightarrow & x \\ i \geq 2 \rightarrow & X[i-1] \end{cases} \\ A[i] = \cdots X[i] \cdots \end{cases}$$

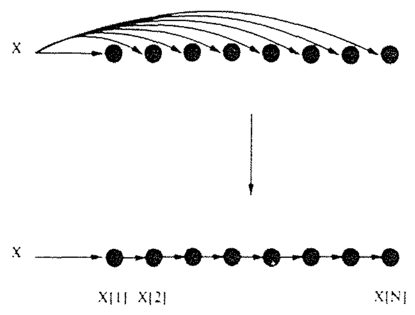


Figure 6.4: Pipelining of a value used in multiple domain points.

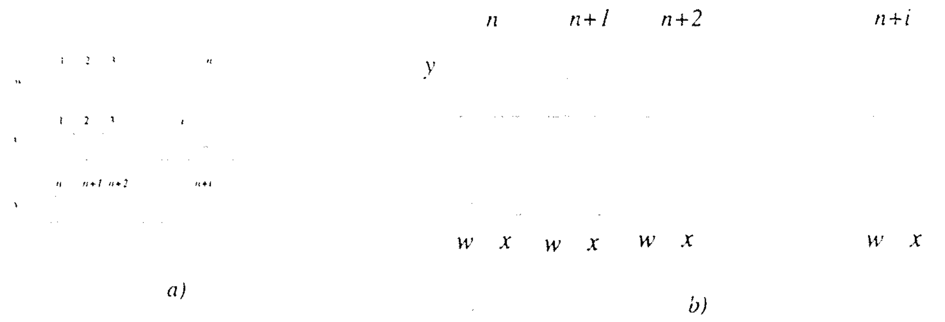


Figure 6.5: a) The initial index domain for each variable of the convolution problem. b) The values required at each index point of y_i.

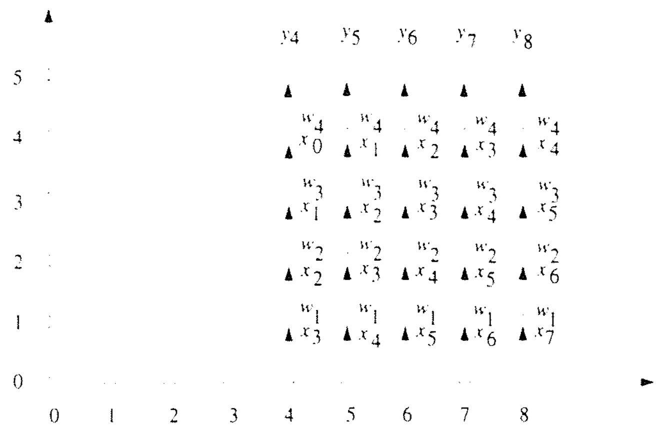
6.4 The Convolution Problem

Let us consider the familiar example of the convolution problem. Given a sequence of values $x_1, x_2, \cdots$, and a vector of weights $w_1, w_2, \cdots, w_N$, the convolution problem is to determine the values y_i for $i \geq N$, given by

$$y_i = \sum_{k=1}^{N} w_k x_{i-k} \quad . \tag{6.1}$$

First, note that we have three different variables, x, y, and w, defined over three different one-dimensional domains, as illustrated in Fig. 6.5a. Considering each index point of y's domain as the computation value for the corresponding y_i, each index point requires the whole set of w's values and a subset of N x values. That is illustrated in Fig. 6.5b. According to our definition of a uniform recurrence equation we require each index point to perform a fixed computation independent of the problem's size. The first transformation that we need is the serialization of the sum operation. We will need several steps before actually we can do that. Let us introduce a new variable Y defined over the two-dimensional space $\{0 \leq i, 0 \leq j \leq N+1\}$. This can be accomplished by a function **AddVariable(** Y, $\{0 \leq i, 0 \leq j \leq N+1\}$ **)**.

Every index point of Y variable is used to compute a single term of the sum for a y_i variable in such a way that the final result for y_i is obtained in $Y(i, N+1)$ for $i \geq N$. In order to do that we need to align variable y with the $N+1$ row of variable Y. This can be accomplished with a function like **Align(** y_i, $Y(i, N+1)$**)**. The serialization of the sum operation can be done, indicating the direction along Y's domain in which we wish the computation be distributed. For our example, let



y_4 y_5 y_6 y_7 y_8
w_4 x_0 x_1 x_2 x_3 x_4
w_3 x_1 x_2 x_3 x_4 x_5
w_2 x_2 x_3 x_4 x_5 x_6
w_1 x_3 x_4 x_5 x_6 x_7
0 1 2 3 4 5
0 1 2 3 4 5 6 7 8

Figure 6.6: Serialization of $\sum_{k=1}^{N} w_k x_{i-k}$ in the convolution problem.

us consider that we distribute the computation of a single y_i along the corresponding column of Y. The call to the function would look like `Serialize(` $\sum_{k=1}^{N} w_k x_{i-k}$ `,` $(0,1)$ `)`, where $(0,1)$ indicates in this case the direction of the serialization. At this point our recurrence equation would look like

$$Y(i,k) = \begin{cases} k = 0 \rightarrow & 0 \\ i \geq N, 1 \leq k \leq N \rightarrow & w_k x_{i-k} + Y(i,k-1) \\ k = N+1 \rightarrow & Y(i,k-1) \end{cases} \tag{6.2}$$

$$i \geq N \rightarrow y_i = Y(i, N+1) \tag{6.3}$$

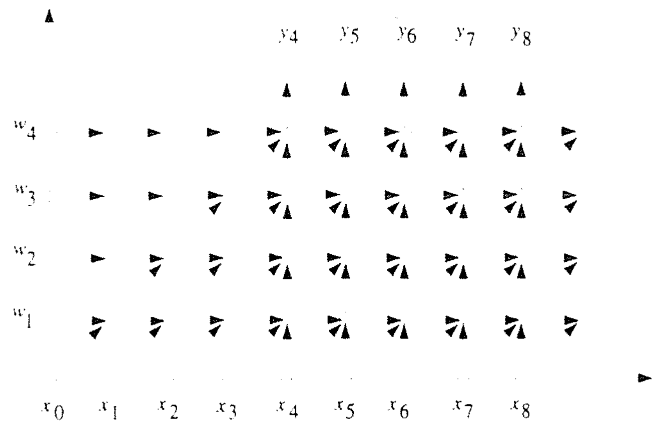
This is illustrated in Fig. 6.6. Note that every index point in the domain $\{i \geq N, 1 \leq K \leq N\}$ actually uses the w and x values indicated in Fig. 6.6, but at this point we have not said how to feed each node with the required values. Observing the pattern shown in the figure, it would be easy to show that we need to pipeline w values along the k-direction and x values along the $(i+1, k+1)$-direction. To dc that we will require introduce two new variables X and W over the same domain that Y and to align x and w values as is shown in Fig. 6.7. Applying the following transformations:

```
AddVariable( $X$, $\{i \geq 0, 0 \leq k \leq N+1\}$ )
Align( $x_i$, $X(i,0)$ )
Pipeline( $x$, (1,1) )
AddVariable( $W$, $\{i \geq 0, 0 \leq k \leq N+1\}$ )
Align( $w_k$, $W(0,k)$ )
Pipeline( $w$, (0,1) )
```

will allow to express our system of recurrence equations in the following manner:

$$W(i,k) = \begin{cases} i = 0, 1 \leq k \leq N \rightarrow & w_k \\ i > 0, 1 \leq k \leq N \rightarrow & W(i-1,k) \end{cases} \tag{6.4}$$

$$X(i,k) = \begin{cases} i \geq 0, k = 0 \rightarrow & x_i \\ i \geq 0, i \leq k \leq N \rightarrow & X(i-1,k-1) \end{cases} \tag{6.5}$$



y_4 y_5 y_6 y_7 y_8
w_4
w_3
w_2
w_1
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8

Figure 6.7: Pipelining x and w values for the convolution problem.

$$Y(i,k) = \begin{cases} i \geq N, k = 0 \rightarrow & 0 \\ i \geq N, 1 \leq k \leq N \rightarrow & W(i,k) \cdot X(i,k) + Y(i,k-1) \\ k = N+1 \rightarrow & Y(i,k-1) \end{cases} \tag{6.6}$$

At this point Eqs. (6.4)-(6.6) represent a system of recurrence equations where all dependencies are uniform. They are represented by the following direction vectors:

$$\delta_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \delta_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \delta_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

It is obvious that we can choose $\lambda = [1 \quad 1]^T$ for the scheduling function and $[0 \quad 1]^T$ for the allocation function. Therefore, we can get the following transformation representing space and time:

$$\begin{bmatrix} t \\ x \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ k \end{bmatrix},$$

The previous transformation maps the dependency vector to the following directions:

$$\delta_1' = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \delta_2' = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \delta_3' = \begin{bmatrix} 2 \\ 1 \end{bmatrix},$$

It should be evident that we have made some important decisions about variable alignment and the directions of serialization and pipelining of variables. In Fig. 6.8 we present another approach where the alignment and serialization correspond to the following steps:

```
AddVariable( $Y$, $\{0 \leq i, 0 \leq j \leq N+1\}$ )
Align( $y_i$, $Y(i+1,0)$)
Serialize( $\sum_{k=1}^{N} w_k x_{i-k}$, (1,-1) )
AddVariable( $X$, $\{i \geq 0, 0 \leq k \leq N+1\}$ )
Align( $x_i$, $X(i,N+1)$ )
Pipeline( $x$, (0,-1) )
AddVariable( $W$, $\{i \geq 0, 0 \leq k \leq N+1\}$ )
Align( $w_k$, $W(0,k)$ )
Pipeline( $w$, (0,1) )
```

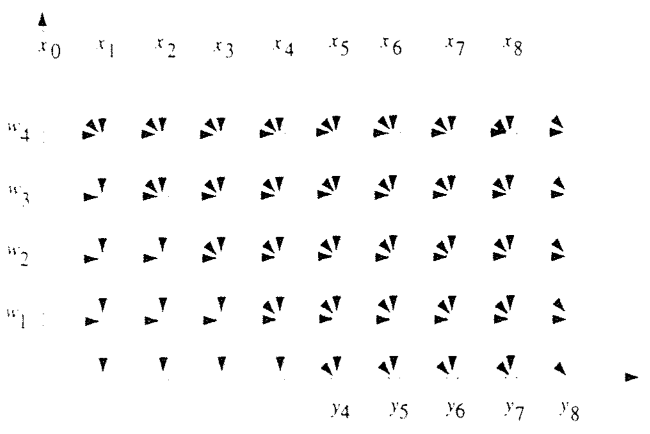

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
w_4
w_3
w_2
w_1
y_4 y_5 y_6 y_7 y_8

Figure 6.8: A different result for the convolution problem changing the positions of alignment and directions of serialization and pipelining.

That sequence of transformations would produce the following system of uniform recurrence equations:

$$W(i,k) = \begin{cases} i = 0, 1 \le k \le N \rightarrow & w_k \\ i > 0, 1 \le k \le N \rightarrow & W(i-1,k) \end{cases} \tag{6.7}$$

$$X(i,k) = \begin{cases} i \ge 0, k = N+1 \rightarrow & x_i \\ i \ge 0, 0 \le k \le N \rightarrow & X(i,k+1) \end{cases} \tag{6.8}$$

$$Y(i,k) = \begin{cases} i \ge N, k = N+1 \rightarrow & 0 \\ i \ge k, 1 \le k \le N \rightarrow & W(i,k) \cdot X(i,k) + Y(i-1,k+1) \\ k = 0 \rightarrow & Y(i-1,k+1) \end{cases} \tag{6.9}$$

For this example the changes in both results are minor, but it should be clear that the decisions taken at different steps would produce different results, and in some cases they would greatly impact the final systolic design.

6.5 The Transitive Closure Problem

In Chapter 2 we introduced the all-pairs shortest path problem. Here we are considering a simplified version of the same algorithm which determines for a given a directed graph $G =< V, E >$ and for each pair of vertices (u, v), if there exists a path joining them. The well known Warshall's algorithm solves this problem based on the following dynamic programming formulation [24]. Let $m_{ij}^{(k)}$ be equal to 1 if there exists a path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \cdots, k\}$. Let it be equal to 0 otherwise. $m_{ij}^{(k)}$ can be computed from the following recurrence equation.

$$m_{ij}^{(k)} = \begin{cases} k = 0 \rightarrow & e_{ij} \\ k \ge 1 \rightarrow & m_{ij}^{(k-1)} \vee (m_{ik}^{(k-1)} \wedge m_{kj}^{(k-1)}) \end{cases} \tag{6.10}$$

where e_{ij} is equal to 1 if there exists an edge from vertex i to vertex j, 0 otherwise. The matrix $M^{(n)} = m_{ij}^{(n)}$ represents the solution to the problem, where n is the number of vertices in the graph.

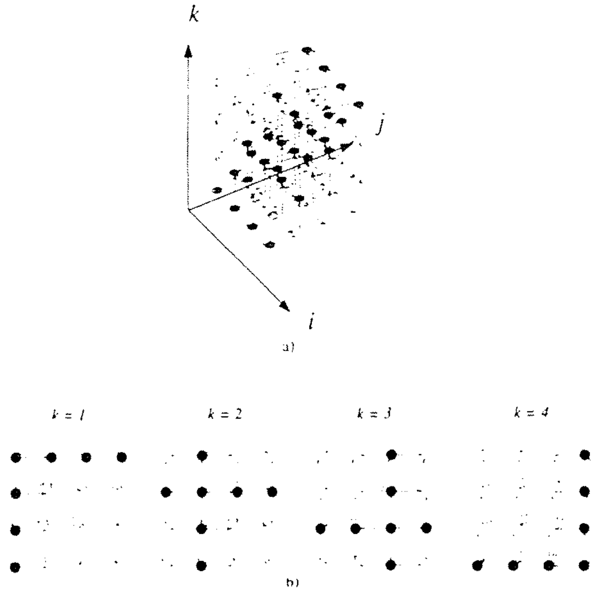


Figure 6.9: a) A 3D view of the dependencies in the transitive closure problem. b) A view by planes of the dependency graph.

According to recurrence 6.10, we have to deal with a three dimensional domain defined by the following inequalities: $1 \leq i \leq n$, $1 \leq j \leq n$, and $1 \leq k \leq n$. If we draw a picture of the whole domain with all of its dependencies, we would obtain a figure like the one shown in Fig. 6.9a (for $n = 4$). We can realize that every index point (k, i, j) depends on $(k-1, i, j)$, but it would be very difficult to have an insight into dependencies in each plane k = constant. Obtaining a view by planes, $k = 1, 2, 3, 4$, like that presented in Fig. 6.9b we can notice immediately the dependencies lying in every plane. It can be observed that in plane k, the kth column is required in every column, and similarly the kth row is required in every row. Following this observation, our first transformation would try to localize these long communications in each plane.

The localization can be done by pipelining the desired values along rows and columns. That involves the introduction of the following variables defined over the same domain.

$$c(k,i,j) = \begin{cases} j < k \rightarrow & c(k,i,j+1) \\ j > k \rightarrow & c(k,i,j-1) \\ j = k \rightarrow & x(k,i,j) \end{cases} \tag{6.11}$$

$$r(k,i,j) = \begin{cases} i < k \rightarrow & r(k,i+1,j) \\ i > k \rightarrow & r(k,i-1,j) \\ i = k \rightarrow & x(k,i,j) \end{cases} \tag{6.12}$$

Introducing these two variables we can express our original recurrence equation in the following way.

$$m(k,i,j) = \begin{cases} k = 0 \rightarrow e_{ij} & \\ 1 \leq k \leq N \rightarrow & m(k-1,i,j) \vee (c(k,i,j) \wedge r(k,i,j)) \end{cases} \tag{6.13}$$

This transformation can lead us to a dependency graph like that shown in Figure 6.10a. At this point we have eliminated the long communications, and we would have a local dependency graph

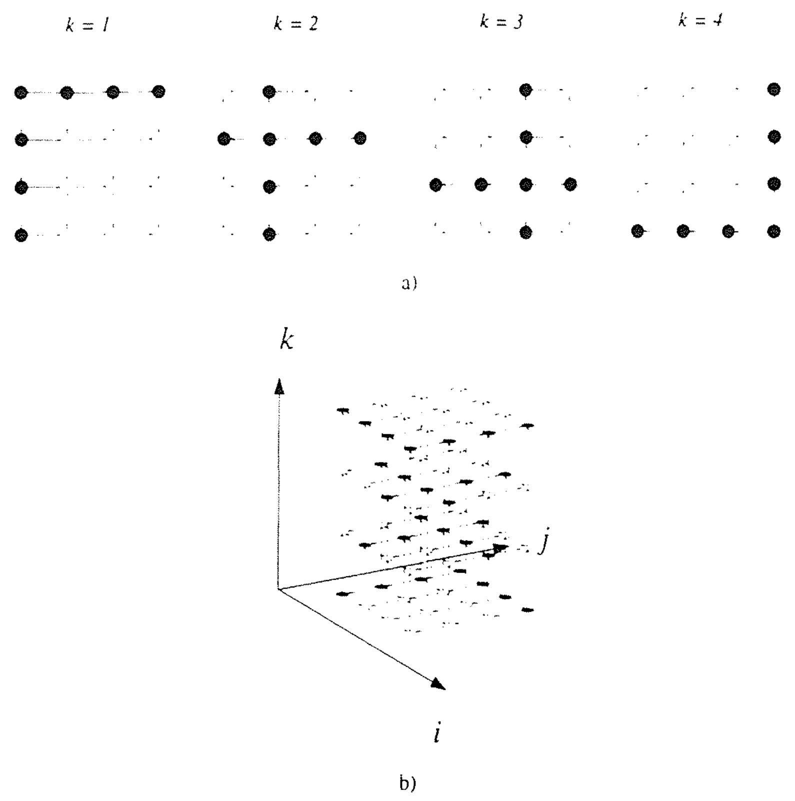


Figure 6.10: Localization communications lying in the same k-plane. a) A view by planes. b) A 3D view.

like shown in Figure 6.10b. However, the direction of dependency vectors varies from plane to plane. To uniformize the dependency directions, we first split the domain in four subdomains, as is shown in Fig. 6.11. Each subdomain can be defined by the following relations:

$$\begin{aligned}
A &= \{(k,i,j) | 1 \le k \le n, 1 \le i < k, 1 \le j < k\} \\
B &= \{(k,i,j) | 1 \le k \le n, 1 < i < k, k \le j \le n\} \\
C &= \{(k,i,j) | 1 \le k \le n, k \le i \le n, 1 \le j < k\} \\
D &= \{(k,i,j) | 1 \le k \le n, k \le i \le n, k \le j \le n\}
\end{aligned}$$

Inside each domain the dependencies are uniform, and they move in the same directions. Let us translate subdomains A, B and C as shown in the Fig. 6.11. This can be expressed by the following transformations:

$$\begin{aligned}
T_A(k,i,j) &= (k, n+i-1, n+j-1) \\
T_B(k,i,j) &= (k, n+i-1, j) \\
T_C(k,i,j) &= (k, i, n+j-1)
\end{aligned}$$

After applying the above transformations we obtain the dependency graph of Fig. 6.12a. Now we can apply the localization step to obtain the dependency graph shown in Fig. 6.12b. We should note that the translations have also moved some dependencies between subsequent planes along k-direction, particularly those k-dependencies lying in the first column and first row of every transformed k-plane. In Fig. 6.13a, we can observe only k-dependencies between two subsequent planes. The transformed set of recurrence equation is the following:

$$c(k,i,j) = \begin{cases} 1 \le k \le N, k \le i \le n+k, k < j \le n+k \to & c(k,i,j-1) \\ 1 \le k \le N, k \le i \le n+k, j = k \to & m(k,i,j) \end{cases} \tag{6.14}$$

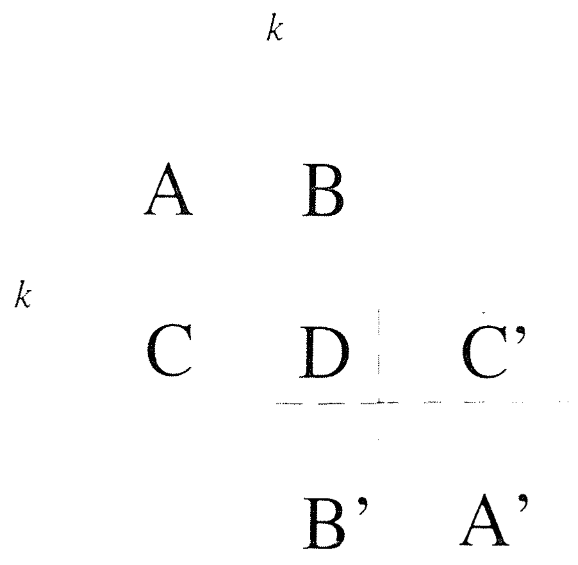


Figure 6.11: Domain splitting and translation proposed to uniformize the dependence directions within each plane shown in Fig. 5a.

$$r(k,i,j) = \begin{cases} 1 \le k \le N, k \le j \le n+k, k < i \le n+k \rightarrow & r(k,i-1,j) \\ 1 \le k \le N, k \le j \le n+k, i = k \rightarrow & m(k,i,j) \end{cases} \tag{6.15}$$

$$m(k,i,j) = \begin{cases} k = 0 \rightarrow \quad e_{ij} \\ 1 \le k \le N \rightarrow \\ \quad \begin{cases} k \le i < n+k-1, k \le j < n+k-1 \rightarrow \\ \qquad m(k-1,i,j) \vee (c(k,i,j) \wedge r(k,i,j)) \\ i = n+k-1, j = n+k-1 \rightarrow \\ \qquad m(k-1,k-1,k-1) \vee (c(k,i,j) \wedge r(k,i,j)) \\ k \le i < n+k-1, j = n+k-1 \rightarrow \\ \qquad m(k-1,i,k-1) \vee (c(k,i,j) \wedge r(k,i,j)) \\ k \le j < n+k-1, i = n+k-1 \rightarrow \\ \qquad m(k-1,k-1,j) \vee (c(k,i,j) \wedge r(k,i,j)) \end{cases} \end{cases} \tag{6.16}$$

Although these long communications shown in Fig. 6.13a are uniform and similar for every pair of subsequent k-planes, we would like to replace them by local communications. It can be seen that the actual m values required at the last column and the last row of plane k are the same as the values propagated along columns and rows in the previous plane through variables c and r. So we can take these values from the nearest index point of the previous k-plane. Furthermore, the value $m(k-1,k-1,k-1)$ required at $m(k,n+k-1,n+k-1)$ never changes. It always is equal to 1, saying that there exists a path from vertex $k-1$ to itself. So, instead of a long propagation of this value we can merely replace this dependency by feeding index points $(k,n+k-1,n+k-1)$ with 1. Both changes can be observed in Fig. 6.13b. We can rewrite Eq. 6.16 in the following way.

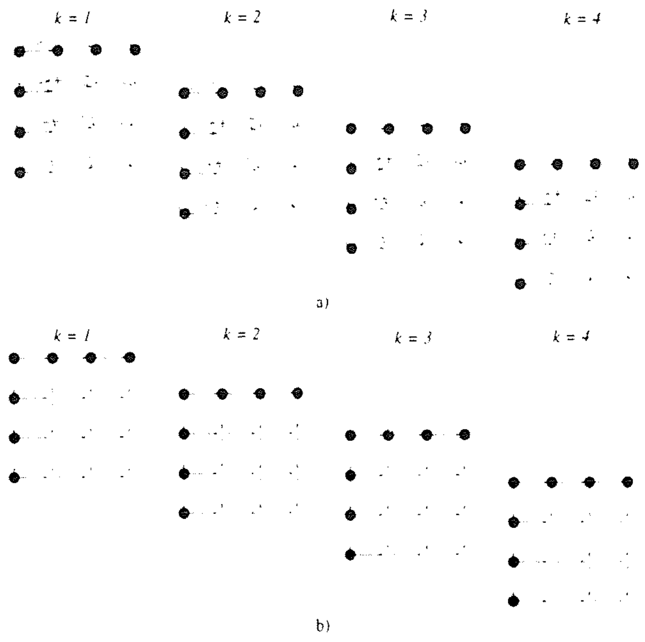


Figure 6.12: Dependency graph obtained first by applying the domain transformation (Fig. 6.11) and then by applying the localization step (Fig. 6.10)
. Each diagram corresponds to a projection to the $i-j$ plane for different value of k as is indicated.

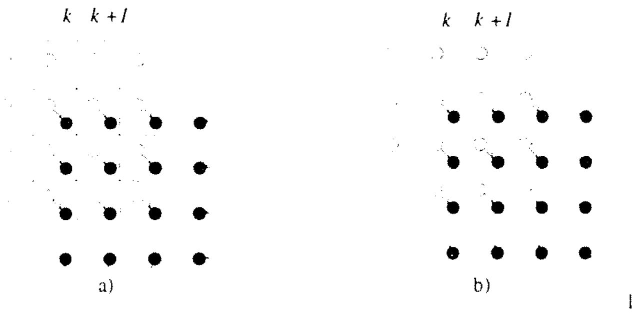


Figure 6.13: a) Changing in dependency vectors between subsequent k-planes. b) Localization long communications (see text).

$$m(k,i,j) = \begin{cases} k=0 \to \quad e_{ij} \\ 1 \le k \le N \to \\ \begin{cases} k \le i < n+k-1, k < j \le n+k-1 \to \\ \quad m(k-1,i,j) \vee (c(k,i,j) \wedge r(k,i,j)) \\ i = n+k-1, j = n+k-1 \to \\ \quad 1 \vee (c(k,i,j) \wedge r(k,i,j)) \\ k \le i < n+k-1, j = n+k-1 \to \\ \quad m(k-1,i,j-1) \vee (c(k,i,j) \wedge r(k,i,j)) \\ k \le j < n+k-1, i = n+k-1 \to \\ \quad m(k-1,i-1,j) \vee (c(k,i,j) \wedge r(k,i,j)) \end{cases} \end{cases} \tag{6.17}$$

In the Fig. 6.14a we show a view of the whole transformed dependency graph. The dependency vectors can be easily recognized. According to the recurrence 6.14, 6.15, and 6.17, they are represented by the following directions $\delta_1 = (1,0,0)^T$, $\delta_2 = (0,1,0)^T$, $\delta_3 = (0,0,1)^T$, $\delta_4 = (1,1,0)^T$, $\delta_4 = (1,0,1)^T$. We can see that a valid allocation function is $a(k,i,j) = (i-k, j-k)$ and hence a

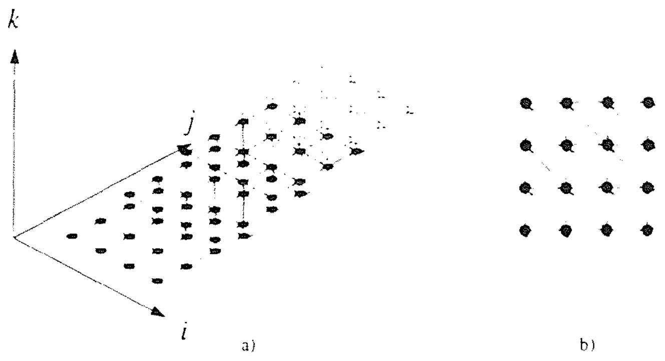


Figure 6.14: a) 3D view of the transformed dependency graph. b) A projection of the domain onto the $(1,1,1)^T$ direction.

valid timing function is $t(k,i,j) = 2k + i + j$. That is described by the following direct and inverse transformations:

$$\begin{bmatrix} t \\ x \\ y \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} k \\ i \\ j \end{bmatrix},$$

$$\begin{bmatrix} k \\ i \\ j \end{bmatrix} = \begin{bmatrix} 1/4 & -1/4 & -1/4 \\ 1/4 & 3/4 & -1/4 \\ 1/4 & -1/4 & 3/4 \end{bmatrix} \begin{bmatrix} t \\ x \\ y \end{bmatrix}$$

Fig. 6.14b shows a projection of the 3D domain into the (i,j) plane along the direction $(1,1,1)$. Applying previous transformations to the dependency vectors we transform them into the following directions $\delta_1' = (2,-1,-1)^T$, $\delta_2' = (1,1,0)^T$, $\delta_3' = (1,0,1)^T$, $\delta_4 = (3,0,-1)^T$, $\delta_5' = (3,-1,0)^T$. Finally, Eqs. 6.14, 6.15, and 6.17 can be transformed to the following system where time and space are made explicit.

$$\bar{C}(t,x,y) = \begin{cases} 4 \le t-(x+y) \le 4N \\ \begin{cases} 0 \le x \le N, 0 < y \le N \rightarrow & \bar{C}(t-1,x,y-1) \\ 0 \le x \le N, y = 0 \rightarrow & \bar{M}(t,x,y) \end{cases} \end{cases} \tag{6.18}$$

$$\bar{R}(t,x,y) = \begin{cases} 4 \le t-(x+y) \le 4N \\ \begin{cases} 0 \le y \le N, 0 < x \le N \rightarrow & \bar{R}(t-1,x-1,y) \\ 0 \le y \le N, x = 0 \rightarrow & \bar{M}(t,x,y) \end{cases} \end{cases} \tag{6.19}$$

$$\bar{M}(t,x,y) = \begin{cases} t = x+y \rightarrow \quad e_{ij} \\ 4 \le t-(x+y) \le 4N \rightarrow \\ \begin{cases} 0 \le x < N-1 \wedge 0 \le y < N-1 \rightarrow \\ \quad \bar{M}(t-2,x+1,y+1) \vee (\bar{C}(t,x,y) \wedge \bar{R}(t,x,y)) \\ x = N-1 \wedge y = N-1 \rightarrow \\ \quad 1 \\ 0 \le x < N-1 \wedge y = N-1 \rightarrow \\ \quad \bar{M}(t-3,x+1,y) \vee (\bar{C}(t,x,y) \wedge \bar{R}(t,x,y)) \\ 0 \le y < N-1 \wedge x = N-1 \rightarrow \\ \quad \bar{M}(t-3,x,y+1) \vee (\bar{C}(t,x,y) \wedge \bar{R}(t,x,y)) \end{cases} \end{cases} \tag{6.20}$$

The set of transformations for the transitive-closure problem can be summarized in the following way:

```
/* Input */
S = Domain( ${(k,i,j)|1 \leq k \leq n, 1 \leq ilen, 1 \leq j \leq n}$ )
AddVariable( $m$, S )
Assign( $m(k,i,j)$, $m(k-1,i,j) \vee (m(k,k,j) \wedge m(k,i,k))$ )
/* Splitting domain S */
A = SubDomain( S, ${(k,i,j)|1 \leq k \leq n, 1 \leq i < k, 1 \leq j < k}$ )
B = SubDomain( S, ${(k,i,j)|1 \leq k \leq n, 1 < i < k, k \leq j \leq n}$ )
C = SubDomain( S, ${(k,i,j)|1 \leq k \leq n, k \leq i \leq n, 1 \leq j < k}$ )
D = SubDomain( S, ${(k,i,j)|1 \leq k \leq n, k \leq i \leq n, k \leq j \leq n}$ )
/* Translation steps */
A' = Translate( A, $(k, n+i-1, n+j-1)$ )
B' = Translate( B, $(k, n+i-1, j)$ )
C' = Translate( C, $(k, i, n+j-1)$ )
S' = DomainUnion( D, A', B', C' )
/* Localization step */
AddVariable( $r$, S' )
Assign( $r(k,k,j)$, $m(k,k,j)$, ${1 \leq k \leq n, k \leq j < n+k}$ )
Pipeline( $m(k,k,j)$, ${1 \leq k \leq n, k \leq j < n+k}$, (0,1,0), $r$ )

AddVariable( $c$, ${1 \leq k \leq n, 1 \leq i < k+n, k \leq j < k+n}$ )
Assign( $c(k,i,k)$, $m(k,k,j)$, ${1 \leq k \leq n, k \leq i < k+n}$ )
Pipeline( $m(k,i,k)$, ${1 \leq k \leq n, k \leq i < k+n}$, (0,0,1), $c$)
/* Elimination of long k-dependencies */
Replace( $m(k-1,k-1,j)$, $r(k-1,i-1,j)$, ${1 < k \leq n, i = k+n-1, k \leq j < k+n}$ )
Replace( $m(k-1,i,k-1)$, $c(k-1,i,j-1)$, ${1 < k \leq n, j = k+n-1, k \leq i < k+n}$ )
Assign( $m(k,i,j)$, 1, ${1 < k \leq n, i = k+n-1, j = k+n-1}$ )
```

6.6 Uniformization of the Gap Problem

In this section we present the uniformization of the Gap problem which was introduced in Chapter 2. Examining the formulation we can observe that to compute a single subproblem, $D[i,j]$, we need $i+j-1$ subproblems, but we can do it in $\max(i,j)$ steps. The goal is to overlap in time the computations of different steps from different problems. Let us introduce the variable $d[i,j,q]$, $0 \leq i,j \leq n$, $0 \leq q \leq \max(i,j)$, to explicitly express the steps to compute $D[i,j]$ in such a way that

$$D[i,j] = d[i,j,\max(i,j)]$$

That implies that the value $D[i,j]$ is ready when $q = \max(i,j)$. The general formulation of $d[i,j,q]$ would be

$$d[i,j,q] = \begin{cases} q = 0 \rightarrow & \textit{initialization} \\ 0 < q \leq \max(i,j) \rightarrow & \min\{d[i,j,q-1], \cdots\} \end{cases} \tag{6.21}$$

where the minimization process is serialized in $\max(i,j)$ steps. Noticing that the minimization operation is associative, we can consider the different $i+j-1$ subproblems in any order. Let us consider first the dependency stated by the subproblem $D[i-1,j-1]$. According to our intended uniformization scheme, it would be available at $d[i-1,j-1,q-1]$. So, in order to avoid long communications, its consideration within the computation of $D[i,j]$ can be done at time $q = \max(i,j)$. So, $d[i,j,q] = \min\{d[i,j,q-1], d[i-1,j-1,q-1]\}$ whenever $q = \max(i,j)$.

To consider the dependencies $D[i,q]$ and $D[q,j]$, let us divide the two-dimensional domain into the three regions given by $i=j$, $i>j$, and $i<j$.

- $i=j$.

  There exist as many rows as columns, so in each step we need to consider a pair of subproblems coming from the same row and the same column. Let us use the variables $d_c[i,j,q]$ and $d_r[i,j,q]$ to indicate the values $D[i,q]$ and $D[q,j]$, respectively, when computing $D[i,j]$. So we can write

$$d[i,j,q] = \begin{cases} q=0 \rightarrow \\ \qquad \min\{d_r[i,j,q]+w(q,j), d_c[i,j,q]+w'(q,i)\} \\ 0<q<i=j \rightarrow \\ \qquad \min\{d[i,j,q-1], d_r[i,j,q]+w(q,j), d_c[i,j,q]+w'(q,i)\} \\ q=i=j \rightarrow \\ \qquad \min\{d[i,j,q-1], d[i-1,j-1,q-1]\} \end{cases} \tag{6.22}$$

  Later on we will describe the formulation for d_r and d_c.

- $i>j$.

  In this case there are more row subproblems than column subproblems. So at any stage q, $0 \leq q < i$, we consider a term $D[q,j]$ represented by $d_c[i,j,q]$. The terms $D[i,q]$ represented by $d_r[i,j,q]$ will be considered after the first $i-j$ steps of the computation. We can write

$$d[i,j,q] = \begin{cases} q=0 \rightarrow \\ \qquad d_r[i,j,q]+w(q,j) \\ 0<q<i-j \rightarrow \\ \qquad \min\{d[i,j,q-1], d_r[i,j,q]+w(q,j)\} \\ i-j \leq q < i \rightarrow \\ \qquad \min\{d[i,j,q-1], d_r[i,j,q]+w(q,j), d_c[i,j,q]+w'(q,i)\} \\ q=i \rightarrow \\ \qquad \min\{d[i,j,q-1], d[i-1,j-1,q-1]\} \end{cases} \tag{6.23}$$

- $i<j$

  This case is the contrast of the previous case. There are more column subproblems than row subproblems. In the first $j-i$ steps only column subproblems are considered. In the following i steps both column and row subproblems are considered. Now we can write

$$d[i,j,q] = \begin{cases} q=0 \rightarrow \\ \qquad d_c[i,j,q]+w'(q,i) \\ 0<q<j-i \rightarrow \\ \qquad \min\{d[i,j,q-1], d_c[i,j,q]+w'(q,i)\} \\ j-i \leq q < j \rightarrow \\ \qquad \min\{d[i,j,q-1], d_r[i,j,q]+w(q,j), d_c[i,j,q]+w'(q,i)\} \\ q=j \rightarrow \\ \qquad \min\{d[i,j,q-1], d[i-1,j-1,q-1]\} \end{cases} \tag{6.24}$$

Let us express now the formulations for d_r and d_c. In Fig. 6.15 we can describe how to propagate the values in the three dimensional domain. Fig. 6.15a only shows the domain. Fig. 6.15b shows the dependencies derived when considering the $D[i-1,j-1]$ terms. In Fig. 6.15c we can observe how to propagate the values along rows. When $i \leq j$, value $D[i,q]$, given by $d[i,q,i]$, will be required in step q when computing $D[i,i]$. This values need to move from (i,q,i) to (i,j,j). Let us move them in the direction $[0,1,-1]$ when $i > j$, and in the direction $[0,1,0]$ when $i \leq j$. These observation give us the following formulation for d_r:

$$d_r[i,j,q] = \begin{cases} 0 \leq j \leq i \leq n \rightarrow & \\ \quad \begin{cases} q = i \rightarrow & d[i,j,q] \\ i-j \leq q < i \rightarrow & d_r[i,j-1,q+1] \end{cases} & \\ 0 \leq i < j \leq n, 0 \leq q < j \rightarrow & d_r[i,j-1,q] \end{cases} \tag{6.25}$$

In Fig. 6.15d) we can observe the dual case for propagating column values. Similar observations can lead to the following formulation for d_c:

$$d_c[i,j,q] = \begin{cases} 0 \leq i \leq j \leq n \rightarrow & \\ \quad \begin{cases} q = j \rightarrow & d[i,j,q] \\ j-i \leq q < j \rightarrow & d_c[i-1,j,q+1] \end{cases} & \\ 0 \leq j < i \leq n, 0 \leq q < i \rightarrow & d_c[i-1,j,q] \end{cases} \tag{6.26}$$

Equations (6.22)-(6.26) describe the complete serialization scheme containing the following dependencies:

$$\delta_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \delta_2 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \delta_3 = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}, \delta_4 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \delta_5 = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}, \delta_6 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

If we express $\lambda = [a,b,c]^T$, by finding a solution of equation $\lambda^T \delta_i > 0$ for δ_i, $1 \leq i \leq 6$ we obtain the following restrictions:
From $\delta_1 \Longrightarrow c > 0$,
From $\delta_2 \Longrightarrow a + b + c > 0$,
From $\delta_3 \Longrightarrow b - c > 0 \Longrightarrow b > c$,
From $\delta_4 \Longrightarrow b > 0$,
From $\delta_5 \Longrightarrow a - c > 0 \Longrightarrow a > c$,
From $\delta_6 \Longrightarrow a > 0$

We can choose $\lambda = [2,2,1]^T$ leading to the timing function $t(i,j,q) = 2i + 2j + q$.
Projecting the domain into the direction $[0,0,1]^T$ we obtain the allocation function $A(i,j,q) = (i,j)$
The direct and the inverse transformation can be formulated in the following matrix form:

$$\begin{bmatrix} t \\ x \\ y \end{bmatrix} = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ q \end{bmatrix}, \quad \begin{bmatrix} i \\ j \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & -2 & -2 \end{bmatrix} \begin{bmatrix} t \\ x \\ y \end{bmatrix}$$

Applying the direct transformation to the dependence vectors we obtain the following mappings:

$$\delta'_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \delta'_2 = \begin{bmatrix} 5 \\ 1 \\ 1 \end{bmatrix}, \delta'_3 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \delta'_4 = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}, \delta'_5 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \delta'_6 = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}$$

To implement the algorithm given by this transformation we need to keep the last five results of variable d and the last two values of each d_r and d_c.

In summary, the transformations to obtain the recurrence equations 6.22 to 6.26 can be described in the following manner:

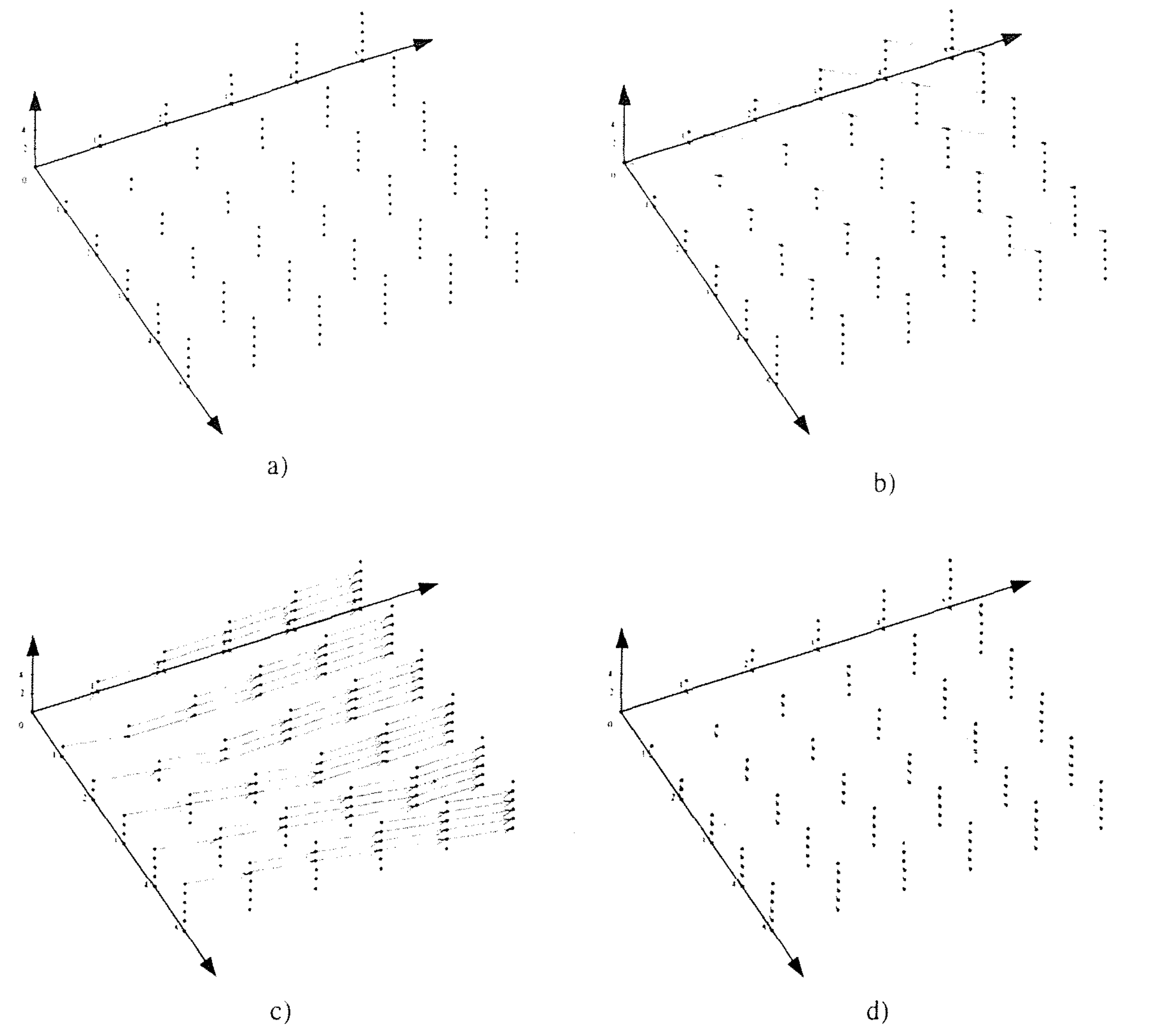


Figure 6.15: Uniformization process of the gap problem. a) The extended 3D domain. b) Dependencies obtained by the D[i-1,j-1] term. c) Propagation of the values along rows. d) Propagation of values along columns

```
S = Domain( $\{(i,j,q)|0 \leq i \leq n, 0 \leq j \leq m, 0 \leq q \leq \max(i,j)\}$ )
AddVariable( $d$, S )
AddVariable( $d_r$, S )
AddVariable( $d_c$, S )

Assign( $d_r(i,j,q)$, $\{j \leq i, q = i\}$, $d(i,j,q)$ )
Pipeline( $d_r(i,j,q)$, $\{j \leq i, q = i\}$, $(0,1,-1)$ )
Pipeline( $d_r(i,j,q)$, $\{i < j, 0 \leq q \leq j\}$, $(0,1,0)$ )

Assign( $d_c(i,j,q)$, $\{i \leq j, q = j\}$, $d(i,j,q)$ )
Pipeline( $d_c(i,j,q)$, $\{i \leq j, q = j\}$, $(1,0,-1)$ )
Pipeline( $d_c(i,j,q)$, $\{j < i, 0 \leq q \leq i\}$, $(1,0,0)$ )

Assign( $d(i,j,q)$, $\{i = j, q = 0\}$,
    $\min(d_r(i,j,q) + w(q,j)$, $d_c(i,j,q) + w'(q,j))$ )

Assign( $d(i,j,q)$, $\{i = j, 0 < q < i\}$,
    $\min(d(i,j,q-1)$, $d_r(i,j,q) + w(q,j)$,
    $d_c(i,j,q) + w'(q,j))$ )
```

```
Assign( $d(i,j,q)$, $\{i = j, q = i\}$,
    min($d(i,j,q-1)$, $d(i-1,j-1,q-1)$) )
Assign( $d(i,j,q)$, $\{i > j, q = 0\}$,
    $d_r(i,j,q) + w(q,j)$ )
Assign( $d(i,j,q)$, $\{i > j, 0 < q < i-j\}$,
    min($d(i,j,q-1)$, $d_r(i,j,q) + w(q,j)$,
Assign( $d(i,j,q)$, $\{i > j, i-j \leq q < i\}$,
    min($d(i,j,q-1)$,
    $d_r(i,j,q) + w(q,j)$, $d_c(i,j,q) + w'(q,i)$) )
Assign( $d(i,j,q)$, $\{i > j, q = i\}$,
    min($d(i,j,q-1)$, $d(i-1,j-1,q-1)$) )
Assign( $d(i,j,q)$, $\{j > i, q = 0\}$,
    $d_c(i,j,q) + w'(q,i)$ )
Assign( $d(i,j,q)$, $\{j > i, 0 < q < j-i\}$,
    min($d(i,j,q-1)$, $d_c(i,j,q) + w'(q,i)$,
Assign( $d(i,j,q)$, $\{j > i, j-i \leq q < j\}$,
    min($d(i,j,q-1)$,
    $d_r(i,j,q) + w(q,j)$, $d_c(i,j,q) + w'(q,i)$) )
Assign( $d(i,j,q)$, $\{j > i, q = j\}$,
    min($d(i,j,q-1)$, $d(i-1,j-1,q-1)$) )
```

6.7 A Tool for Manipulating Recurrence Equations

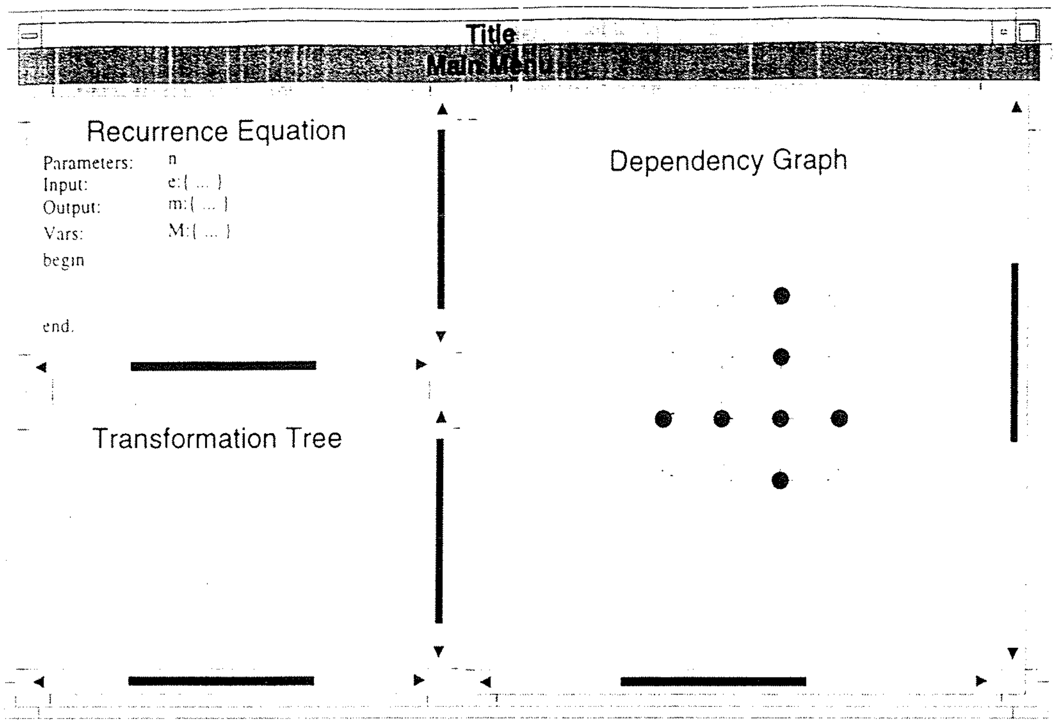
In this section we specify an interactive tool to manipulate recurrence equations. The goal of the tool is to let the user explore different transformations before selecting a sequence of steps to transform the input recurrence equations into output uniform recurrence equations. The tool relies on a graphical interface to display the dependency graph of a problem. By letting the user specify different views of the dependency graph, the tool should help him derive the right set and order of transformations. That feature should be helpful for users with little or no experience dealing with 2D or 3D problems.

The major purpose of the tool is to help the user transform a recurrence equation into a set of uniform recurrence equations. This process involves deriving an ordered set of suitable transformations. To facilitate that process the tool must accomplish three major objectives. First, it must let the user specify the transformations in a suitable way. Secondly, it must let the user keep track of the global state of the transformation process. Finally, the tool must provide the user with convenient feedback from each decision he takes. Let us explore now how to fulfill these goals.

The tool should have predefined a set of basic transformations suitable to implement the processes of alignment, localization and serialization of computations. Furthermore, the tool has to deal with the transformation of domains (Figure 6.16a).

In the process of exploring different approaches, the user builds a tree of transformations where the correct sequence is a path from the root to one of the leaves as it is shown in Figure 6.16b. This transformation tree represents the different transformations applied by the user during the process of uniformization. One of its nodes should represent the current state of the transformation process. The tool should have facilities to backtrack to a past state and to continue the process from that state.

One way to assist the user in the transformation process is to provide him with a graphical representation of the problem at hand. That would be possible only for 2D and 3D problems. Showing the existing dependencies of the problem usually will help to see which dependencies should be localized and which computations should be serialized. How to do that is the user's responsibility. So, the tool should have facilities to obtain different views of the problem's dependency graph. That



```
Parameters:   n
Input:        e:{ ... }
Output:       m:{ ... }
Vars:         M:{ ... }
begin

end.
```

Figure 6.17: The main screen of the tool to manipulate recurrence equations.

a predefined plane: $i-j$, $i-k$, or $k-j$. Also we might choose a subdomain of the recurrence equation and a subset of the dependencies to show in the current view of the dependency graph. Some additional features such as highlighting source or target index points and highlighting uniform or nonuniform dependencies can be of great help to the user. Depending on the complexity of the dependency graph, we can supply that information using dialogs or direct manipulation.

In Figure 6.18 we illustrate the direct manipulation actions to specify the partitioning of the domain for the transitive closure problem. The purpose is to move all the index points to make the dependency arcs point along the same direction. By enclosing the index points within region A we can group them and translate them to the position shown in Figure 6.18b. Then, we can apply the same grouping, clicking and dragging operations to translate index point inside regions B and C to obtain Figures 6.18c) and 6.18d), respectively. Note that the dependency arcs should move as we translate the index points.

In summary, the direct manipulation features of a mouse such as pointing, grouping, clicking and dragging can be used to select index points, to select dependencies, to specify directions or translations, to change source or destination of dependency arcs, and to redefine a new source value. Complimentary information required to completely specify a transformation can be requested by input dialogs.

6.8 Conclusions

We have exemplified the uniformization process of recurrence equations. The goal is to transform a general recurrence equation into a system of uniform recurrence equations where dependencies are constant and fixed. The intermediate steps work over domains or computations to obtain new representations. Serialization, alignment and localization are key steps in uniformizing a recurrence equation. Good skill in manipulating polyhedral domains is suitable to perform such tasks. By combining facilities to derive transformations with facilities to visualize domains and dependencies we can derive sets of transformations to achieve, whenever it is possible, uniform representations.

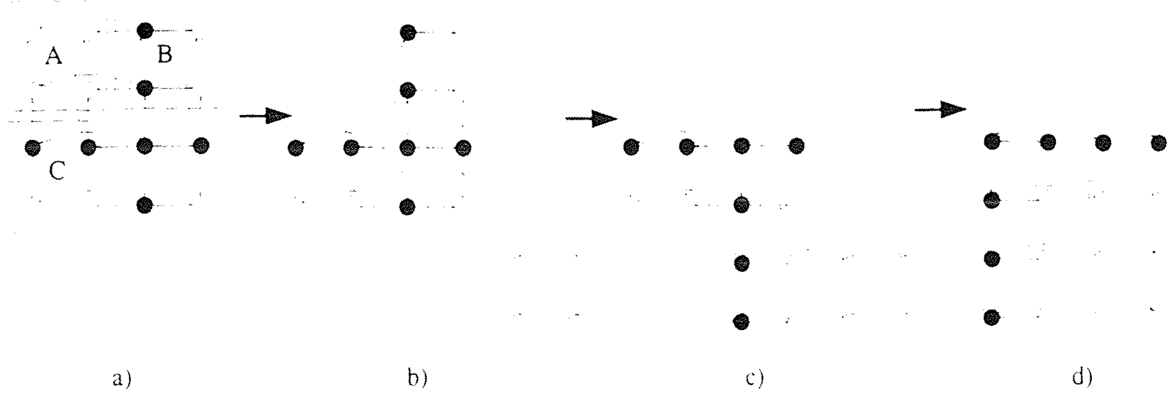


Figure 6.18: An example partitioning and translation transformations using direct manipulation.

We have illustrated the uniformization process for the three problems: convolution, transitive closure and gap problems. To the best of our knowledge no previous uniform representation for the gap problem has been found. We also introduced a set of functions to perform the required transformations.

We have outlined the design of a tool for manipulating recurrence equations. It is intended to transform, with the user's aid, a general recurrence equation into a system of uniform recurrence equations suitable for applying systolic processing. This tool would consist of a predefined set of transformations. Each transformation can manipulate either or both the domain and the computation of a recurrence equation. We believe the tool's capabilities to backtrack and to obtain graphical representations of the problem at hand would be of great help during the uniformization process.

Chapter 7

Generating VHDL Behavioral Models

7.1 Introduction

Space-time descriptions have been used as the final representation for a systolic array. Since systolic arrays are intended to be implemented on hardware, it is required to generate a circuit design from space-time descriptions. VHDL can be used as a target language for that purpose, given that it is a standard language and a variety of tools exists for circuit implementation from it. In generating circuit designs from space-time representations, we need to make several decisions. For example, we need to specify parameters for implementation, we have to design a generic processor for the array and we have to replicate it indicating interconnections between different instances. Finally, we must specify control signals to activate each processor in a synchronous way. In this chapter we discuss the main issues in generating circuit designs in VHDL from space-time representations and we present some approaches to the problem.

In section 7.2 we outline the digital design process. In section 7.3, we briefly review VHDL features. In section 7.4, we define space-time representations and we introduce some examples. In section 7.5, we discuss the main issues in generating VHDL behavioral descriptions from space-time representations. Important aspects about control signal generation, port configuration and processor description are reviewed in that section. A case study appears in section 7.6.

7.2 Digital Design Process Review

The design process of a digital system consists of transforming a given functional specification to a suitable level for its fabrication or construction [48]. It usually can be described as an abstraction process from a high-level specification to a low-level one as it is shown in Figure 7.1. The design process usually starts from an idea described at the behavioral level, in which a system is described by its input-output response. That behavior is translated to a register transfer level (RTL) in which a set of resources are specified, a mapping function indicates which behavior is executed by each resource and a scheduling strategy defines the order in which the operations of the behavior are performed. This level is usually described as a set of registers and a control device which are interconnected by one or several buses. Then, a RTL description is transformed to a logic level in which a number of gates are specified as well as their interconnections by wires. It is usually called a netlist or a gate wirelist. From the logic level a transistor list or a layout description can be obtained which finally produces a circuit that performs the initial description.

There exist a number of tools which assist humans in making that transformation process. Simulators and synthesizers are among the most used. Simulators perform the design at the level provided by the design entry. They usually require input vectors test patterns to do the simulation. Synthesizers are automatic tools for transforming one design at one level of abstraction to a lower level.

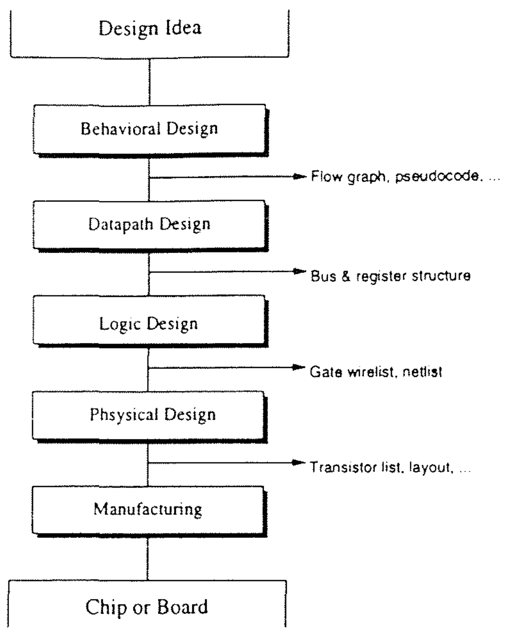


Figure 7.1: A typical abstraction hierarchy of the digital design process.

We would like to have a synthesizer for transforming a behavioral description to a layout. Since that work involves many formidable tasks such as optimization, placement and routing, most current tools transform a RTL description to gate wirelists [32]. There have been distinguished two types of synthesizers: high-level and low-level. While high-level synthesizers take as input a behavioral description, a low-level one translates a gate-level description to a layout circuit.

In summary, design is a series of transformations from one representation of a system to another until a representation exists that can be fabricated. Synthesis is the process of transforming one representation in the design abstraction hierarchy to another. There exist two types of descriptions: behavioral and structural. A behavioral description is represented by an input-output response. It can be done algorithmically or by a dataflow representation. On the other hand, a structural representation is described in terms of an interconnection of more primitive components. At each level of the design hierarchy some procedures are necessary to do design entry, simulation, synthesis and test design.

7.3 VHDL Review

The VHSIC Hardware Description Language, VHDL, supports modeling and simulation of circuits at all stages of the design process. VHDL allows to do synthesis, hardware testing and timing analysis of a circuit. In VHDL, a circuit component is represented as an entity which may be associated with various alternative architectures. Typically, an architecture may either specify an abstract behavioral description of a device, or provide a concrete structural definition in terms of simpler components (see Figure 7.2). The equivalence of architectures may be confirmed through comparative simulations.

A VHDL environment consists of an analyzer which processes the input provided by a text editor in VHDL syntax. It checks for syntax errors and when the input-code is error-free, translates it to an intermediate format which is then linked with one or several libraries by a library subsystem. The output can be used for simulation, synthesis, or testing, by a set of tools that take the intermediate representation as input (see Figure 7.3).

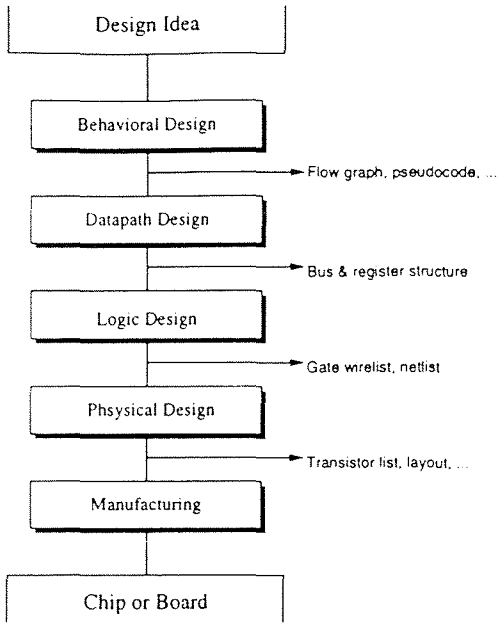


Figure 7.1: A typical abstraction hierarchy of the digital design process.

We would like to have a synthesizer for transforming a behavioral description to a layout. Since that work involves many formidable tasks such as optimization, placement and routing, most current tools transform a RTL description to gate wirelists [32]. There have been distinguished two types of synthesizers: high-level and low-level. While high-level synthesizers take as input a behavioral description, a low-level one translates a gate-level description to a layout circuit.

In summary, design is a series of transformations from one representation of a system to another until a representation exists that can be fabricated. Synthesis is the process of transforming one representation in the design abstraction hierarchy to another. There exist two types of descriptions: behavioral and structural. A behavioral description is represented by an input-output response. It can be done algorithmically or by a dataflow representation. On the other hand, a structural representation is described in terms of an interconnection of more primitive components. At each level of the design hierarchy some procedures are necessary to do design entry, simulation, synthesis and test design.

7.3 VHDL Review

The VHSIC Hardware Description Language, VHDL, supports modeling and simulation of circuits at all stages of the design process. VHDL allows to do synthesis, hardware testing and timing analysis of a circuit. In VHDL, a circuit component is represented as an entity which may be associated with various alternative architectures. Typically, an architecture may either specify an abstract behavioral description of a device, or provide a concrete structural definition in terms of simpler components (see Figure 7.2). The equivalence of architectures may be confirmed through comparative simulations.

A VHDL environment consists of an analyzer which processes the input provided by a text editor in VHDL syntax. It checks for syntax errors and when the input-code is error-free, translates it to an intermediate format which is then linked with one or several libraries by a library subsystem. The output can be used for simulation, synthesis, or testing, by a set of tools that take the intermediate representation as input (see Figure 7.3).

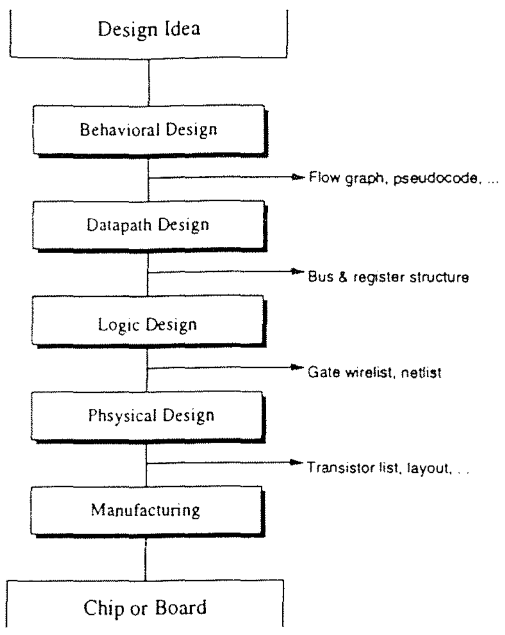


Figure 7.1: A typical abstraction hierarchy of the digital design process.

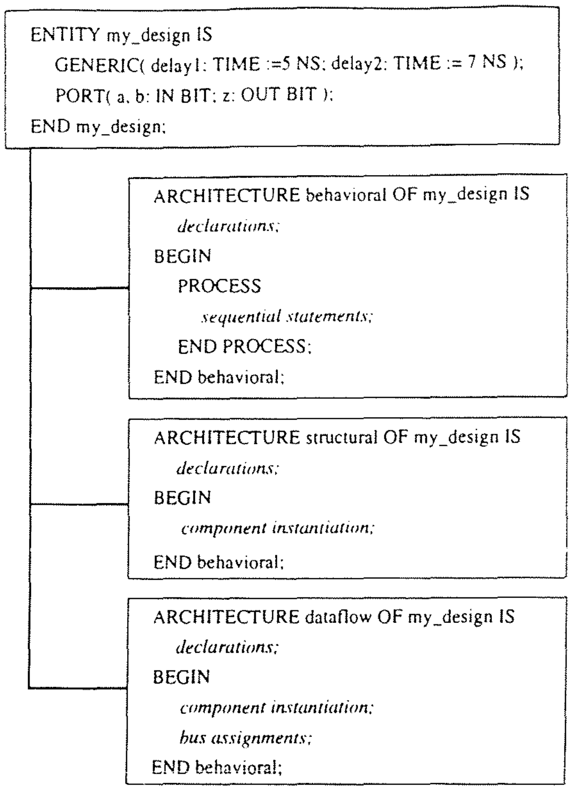
We would like to have a synthesizer for transforming a behavioral description to a layout. Since that work involves many formidable tasks such as optimization, placement and routing, most current tools transform a RTL description to gate wirelists [32]. There have been distinguished two types of synthesizers: high-level and low-level. While high-level synthesizers take as input a behavioral description, a low-level one translates a gate-level description to a layout circuit.

In summary, design is a series of transformations from one representation of a system to another until a representation exists that can be fabricated. Synthesis is the process of transforming one representation in the design abstraction hierarchy to another. There exist two types of descriptions: behavioral and structural. A behavioral description is represented by an input-output response. It can be done algorithmically or by a dataflow representation. On the other hand, a structural representation is described in terms of an interconnection of more primitive components. At each level of the design hierarchy some procedures are necessary to do design entry, simulation, synthesis and test design.

7.3 VHDL Review

The VHSIC Hardware Description Language, VHDL, supports modeling and simulation of circuits at all stages of the design process. VHDL allows to do synthesis, hardware testing and timing analysis of a circuit. In VHDL, a circuit component is represented as an entity which may be associated with various alternative architectures. Typically, an architecture may either specify an abstract behavioral description of a device, or provide a concrete structural definition in terms of simpler components (see Figure 7.2). The equivalence of architectures may be confirmed through comparative simulations.

A VHDL environment consists of an analyzer which processes the input provided by a text editor in VHDL syntax. It checks for syntax errors and when the input-code is error-free, translates it to an intermediate format which is then linked with one or several libraries by a library subsystem. The output can be used for simulation, synthesis, or testing, by a set of tools that take the intermediate representation as input (see Figure 7.3).



```
ENTITY my_design IS
  GENERIC( delay1: TIME :=5 NS; delay2: TIME := 7 NS );
  PORT( a, b: IN BIT; z: OUT BIT );
END my_design;

ARCHITECTURE behavioral OF my_design IS
  declarations;
BEGIN
  PROCESS
    sequential statements;
  END PROCESS;
END behavioral;

ARCHITECTURE structural OF my_design IS
  declarations;
BEGIN
  component instantiation;
END behavioral;

ARCHITECTURE dataflow OF my_design IS
  declarations;
BEGIN
  component instantiation;
  bus assignments;
END behavioral;
```

Figure 7.2: A typical VHDL with alternative descriptions.

VHDL is a concurrent language which allows a multilevel description of a system. It includes support for concurrent management of signal assignments as well as the usual sequential descriptions. VHDL design usually are generic which later on are bound to specific libraries. VHDL can manage packages, subprograms and procedures. It is a strongly typed language whose basic type is a signal which is a variable with a time component. VHDL allows concurrent assignment of signals by handling delay assignments.

Usually the VHDL simulation system is an event-driven simulator which evaluates a circuit only when events occur. The simulator keeps track of two different times: a simulation time and a real time. The events are recorded by a time queue processor which keeps track of real time in a queue time. It generates events to be handled by a signal tracer. When the delay component of an event has reached zero value, it activates a signal change which is processed by a process executor which, in turn, schedules new events to be tracked (See Figure 7.4).

7.4 Space-Time Descriptions

In Chapter 4 we introduced recurrence equations and space-time descriptions. In this section we will further elaborate the definition of space-time descriptions to make some observations about their structure. Let $X, Y, Z, \ldots$ be *identifiers* (also called *variables* or *functions* as we will see later).

A *space-time description* is a uniform recurrence equation defining an identifier X over a domain $\mathcal{D}_i^X$ of the form

$$X(\mathbf{z}) = g(Y_1(\mathbf{z} - \delta_1), \ldots, Y_m(\mathbf{z} - \delta_m)) \quad \forall \mathbf{z} \in \mathcal{D}_i^X \tag{7.1}$$

where

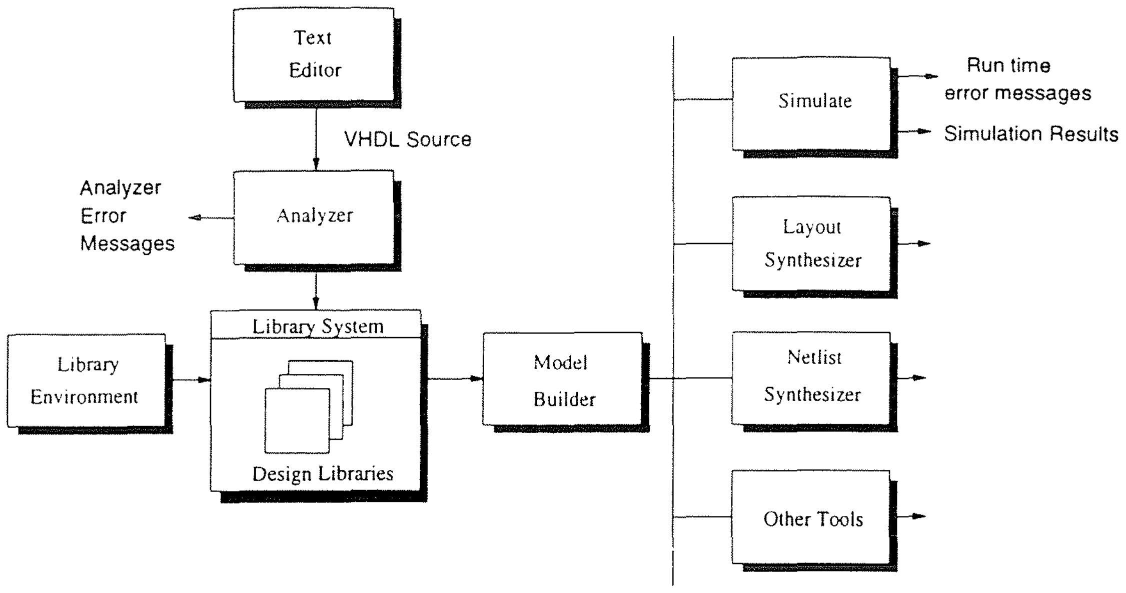


Figure 7.3: A VHDL working environment.

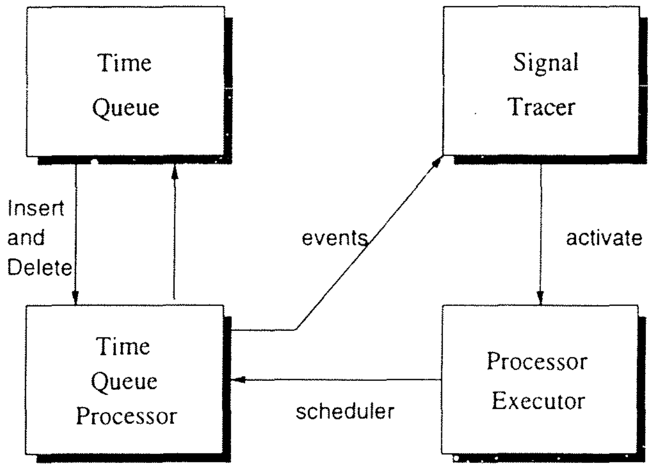


Figure 7.4: A typical VHDL simulator.

- $\mathcal{D}_i^X = T \times S^{n-1}$ is the *domain of definition* of the equation which is a n-dimensional space whose one dimension is designated as *time* and the other $(n-1)$ dimensions are designated as *space*. The space describes an $(n-1)$-dimensional array of points which are interpreted as virtual processors. If two or more recurrence equations all have the same identifier X on the left-hand-side of the equation, then their respective domains of definitions are disjoint.

- The points $\mathbf{z}$ of the domain are of the form $\mathbf{z} = (t, \mathbf{s})$ where t is a 1-dimensional value identified with time and $\mathbf{s}$ is an $(n-1)$-dimensional value which is identified as space.

- The expressions $\mathbf{z} - \boldsymbol{\delta}_j$, $1 \leq j \leq m$, are considered as *dependency mapping functions* (also called *index mapping functions*) which map $\mathbf{z}$ in $\mathcal{D}_i^X$ to $\mathbf{z} - \boldsymbol{\delta}_j, 1 \leq j \leq m$.

- $Y_j, 1 \leq j \leq m$, are identifiers which can be considered as inputs or can be defined by their own space-time descriptions.

- g_i is a strict single-valued function defining the right-hand-side of the equation.

According the definition, the constant vectors $\delta_1, \delta_2, \dots, \delta_m$ are defined as communications between virtual processors of the array. They are n-dimensional values. One of them indicates delays, and the $n-1$ remaining values indicate communication to a neighbor processor of $\mathbf{z}$ in a given direction.

Similarly to recurrence equations, domains of definition for space-time descriptions of an identifier represent a partition of the domain of the identifier. Since all identifiers are frequently defined over the same domain, we will denote it simply as $\mathcal{D}$.

Usual domains are n-dimensional integer polyhedrons which were defined in Chapter 5.

We can group all space-time description in whose left-hand-side appears the same variable or identifier and write the definition of a variable in the following form

$$X(\mathbf{z}) = \begin{cases} \forall \mathbf{z} \in \mathcal{D}_1^X \rightarrow & g_1(Y_{11}(\mathbf{z}-\delta_{11}), \dots, Y_{1m_1}(\mathbf{z}-\delta_{1m})) \\ \dots & \\ \forall \mathbf{z} \in \mathcal{D}_n^X \rightarrow & g_n(Y_{n1}(\mathbf{z}-\delta_{n1}), \dots, Y_{nm_1}(\mathbf{z}-\delta_{nm})) \end{cases} \tag{7.2}$$

It is common to write the conditions $\forall \mathbf{z} \in \mathcal{D}_i^X$, $1 \leq i \leq n$, as predicates, $p_i^X(\mathbf{z})$, which take true or false values (1 or 0) and $\sum_i p_i^X(\mathbf{z}) = 1$, i. e., one and only one of the predicates, p_i^X, takes a true value in a point $\mathbf{z}$. Then, we can rewrite Equation 7.2 as

$$X(\mathbf{z}) = \begin{cases} p_1^X(\mathbf{z}) \rightarrow & g_1(Y_{11}(\mathbf{z}-\delta_{11}), \dots, Y_{1m_1}((\mathbf{z}-\delta_{1m})) \\ \dots & \\ p_n^X(\mathbf{z}) \rightarrow & g_n(Y_{n1}(\mathbf{z}-\delta_{n1}), \dots, Y_{nm_1}((\mathbf{z}-\delta_{nm})) \end{cases} \tag{7.3}$$

indicating that if $p_i^X(\mathbf{z})$ takes a true value, then $X(\mathbf{z})$ is evaluated to the corresponding g_i function.

As we will see in the next section, in most of the cases the predicates can be written as a conjunction

$$p_i = P_i^1 \wedge P_i^2 \wedge \cdots P_i^l \tag{7.4}$$

of predicates P_i^j taking one of the following forms

$$\begin{array}{l} p_i^j(x_1, x_2, \cdots, x_{n-1}) < t \\ p_i^j(x_1, x_2, \cdots, x_{n-1}) = t \\ p_i^j(x_1, x_2, \cdots, x_{n-1}) > t \end{array} \tag{7.5}$$

Each p_i^j is seen as a control expression involving only space indexes, x_i. Usually, it evaluates to a nonnegative rational value and the expression consists only of rational expressions and terms to the first power.

Usually, not all δ_{ij}, $1 \leq i \leq n, 1 \leq j \leq m$ are different. Then, we will refer to them simply as the set of dependencies $\{\delta_1, \dots, \delta_d\}$. Let us examine the different possibilities of $\mathbf{z} - \delta_k$, $1 \leq k \leq d$.

1. $Y \neq X$. g_i is referring to different identifiers. That exhibits the following possibilities:
   (a) $\mathbf{z} - \delta = (t, x_1, x_2, \cdots, x_{n-1})$. This represents an access to a different identifier in the same domain point at the current time.
   (b) $\mathbf{z} - \delta = (t - \Delta t, x_1, x_2, \cdots, x_{n-1})$. A reading to a value taken by a different identifier in the same domain point at previous time. Δt indicates the number of clock cycles previous to the current one in which the desired value of the variable was produced.
   (c) $\mathbf{z} - \delta = (t - \Delta t, x_1 - \Delta x_1, x_2 - \Delta x_2, \cdots, x_{n-1} - \Delta x_{n-1})$. A reading to a value taken by a different identifier in a neighbor domain point at previous time. $(\Delta x_1, \Delta x_2, \cdots, \Delta x_{n-1})$ indicates the direction and displacement in which the neighbor point is located. Note that $\Delta t > 0$, since we cannot access values at different space point at the time in which they are produced.

2. $Y = X$. That means that g_i is referring to the same identifier.

   (a) $\mathbf{z} - \boldsymbol{\delta} = (t - \Delta t, x_1, x_2, \cdots, x_{n-1})$. A reading to a value taken by the identifier in the same domain point at previous time. Δ_t indicates the number of clock cycles previous to the current one in which the desired value was produced.

   (b) $\mathbf{z} - \boldsymbol{\delta} = (t - \Delta t, x_1 - \Delta x_1, x_2 - \Delta x_2, \cdots, x_{n-1} - \Delta x_{n-1})$. A reading to a value taken by the same identifier in a neighbor domain point at previous time. $(\Delta x_1, \Delta x_2, \cdots, \Delta x_{n-1})$ also indicates the direction and displacement in which the neighbor point is located. For the same reasons as above, $\Delta t > 0$

We can make some observations about space-time representations. First, Δt must be greater than or equal to 0. When $\Delta t > 0$ we refer the access with delay Δt. We cannot access variables in a different domain point with $\Delta t = 0$. When $\Delta x = (\Delta x_1, \Delta x_2, \ldots, \Delta x_{n-1}) \neq 0^T$ we consider there is a communication to a neighbor processor located with displacement Δx.

7.5 Important Issues in Generating Behavioral Descriptions

Once we have made some remarks about space-time representation we will focus on the main issues in generating behavioral descriptions from them in VHDL. Each following subsection will consider a different aspect. The first one is of importance to simplify the understanding of remaining sections.

7.5.1 Control Signal Handling

Predicates appearing in a given space-time description indicate the control of the functionality and timing of the processors. Since we intend to implement circuits, dedicating hardware to perform those could be expensive. However, as we pointed out before, they usually involve only terms to the first power and this makes them susceptible of some optimizations. A better design could be obtained by replacing the expensive computations of a predicate by transferring one-bit control signal, as illustrated by [18, 39]. For a predicate that is independent of t there is no concern, since it can be hardwired into the design. For any predicate that is dependent on t, it must be substituted by one that is independent of t. Since a communication always moves in space and takes time to complete, it can be used to "compute" expressions of the space-time indices in a time-variant predicate.

A *control expression* is an expression $p(x_1, x_2, \cdots, x_{n-1})$ of space indices x_i, $1 \leq i < n$ which evaluates to a nonnegative rational value and the expression consists only of rational coefficients and terms to the first power.

Proposition 1 *Suppose a control expression $p(x_1, x_2, \cdots, x_{n-1})$ is strictly monotonic increasing in some space index x_i, that means, if $x'_i < x_i$, then $p(x_1, \cdots, x'_i, \cdots, x_{n-1}) < p(x_1, \cdots, x_i, \cdots, x_{n-1})$. Then, for each x_i that accomplishes the previous restriction, the time dependence of the control expression, defined as*

$$dt(x_i) = p(x_1, \cdots, x_i, \cdots, x_{n-1}) - p(x_1, \cdots, x_i - 1, \cdots, x_{n-1}) \tag{7.6}$$

has a positive constant value.

The previous proposition can be easily verified since p involves terms to the first power. A similar proposition is valid when p is strictly monotonic decreasing.

Proposition 2 *Suppose a control expression $p(x_1, x_2, \cdots, x_{n-1})$ is strictly monotonic decreasing in some space index x_i, that means, if $x'_i < x_i$, then $p(x_1, \cdots, x'_i, \cdots, x_{n-1}) > p(x_1, \cdots, x_i, \cdots, x_{n-1})$. Then, for each x_i that accomplishes the previous restriction, the time dependence of the control expression, defined as*

$$dt(x_i) = p(x_1, \cdots, x_i, \cdots, x_{n-1}) - p(x_1, \cdots, x_i + 1, \cdots, x_{n-1}) \tag{7.7}$$

has a positive value which is constant.

Given the above propositions we can state the following theorem.

Theorem 1 *Time-variant predicates of the form*

$$\begin{aligned} t &< p(x_1, x_2, \cdots, x_{n-1}) \\ t &= p(x_1, x_2, \cdots, x_{n-1}), \text{ and} \\ t &> p(x_1, x_2, \cdots, x_{n-1}) \end{aligned}$$

where $p(x_1, x_2, \cdots, x_{n-1})$ is a strictly monotonic increasing control expression, can be implemented by predicates

$$q(x_1, x_2, \cdots, x_{n-1}, t) = s$$

where,

$$s = \begin{cases} 0 & for \quad t < p(x_1, x_2, \cdots, x_{n-1}) \\ 1 & for \quad t = p(x_1, x_2, \cdots, x_{n-1}) \\ 2 & for \quad t > p(x_1, x_2, \cdots, x_{n-1}) \end{cases}$$

The control stream $q(x_1, x_2, \cdots, x_{n-1}, t)$ is defined by

$$q(t, x_1, x_2, \cdots, x_{n-1}) = \begin{cases} p(x_1, x_2, \cdots, x_{n-1}) = 0 \rightarrow \\ \qquad \begin{cases} t < 0 \rightarrow & 0 \\ t = 0 \rightarrow & 1 \\ t > 0 \rightarrow & 2 \end{cases} \\ p(x_1, x_2, \cdots, x_{n-1}) > 0 \rightarrow \\ \qquad q(t - dt(x_i), x_1, x_2, \cdots, x_{i-1}, \cdots, x_{n-1}) \end{cases} \tag{7.8}$$

in which all predicates, except for the switch-on predicates $t < 0$, $t = 0$, and $t > 0$ are time-invariant, and $dt(x_i)$ is the time dependency of control expression $p(x_1, x_2, \cdots, x_{n-1})$.

Proof. First, let us make the following observation. From 7.6 we can write

$$p(x_1, \cdots, x_i, \cdots, x_{n-1}) = p(x_1, \cdots, x_i - 1, \cdots, x_{n-1}) + dt(x_i) \tag{7.9}$$

Recursively applying 7.9 over $p(x_1, \cdots, x_i - 1, \cdots, x_{n-1})$ we can find the following expression

$$p(x_1, \cdots, x_i, \cdots, x_{n-1}) = p(x_1, \cdots, x_i - k, \cdots, x_{n-1}) + kdt(x_i), k > 0 \tag{7.10}$$

Since $p(x_1, \cdots, x_i, \cdots, x_{n-1})$ is strictly monotonic

$$p(x_1, \cdots, x_i, \cdots, x_{n-1}) > p(x_1, \cdots, x_i - 1, \cdots, x_{n-1}) > \cdots > p(x_1, \cdots, x_i - k, \cdots, x_{n-1}) \geq 0 \tag{7.11}$$

Without losing generality, we can assume that there exists j such that

$$p(x_1, \cdots, x_i - j, \cdots, x_{n-1}) = 0 \tag{7.12}$$

Let us focus now on the part of the theorem related to the case $t < p(x_1, \cdots, x_i, \cdots, x_{n-1})$. The proof of two other cases is similar.

We want to prove that

$$t < p(x_1, \cdots, x_{n-1}) \Longleftrightarrow q(t, x_1, x_2, \cdots, x_{n-1}) = 0 \tag{7.13}$$

Let us separate the case when we already know that $p(x_1, \cdots, x_{n-1}) = 0$ from the case when $p(x_1, \cdots, x_{n-1}) > 0$

1. $p(x_1, \cdots, x_{n-1}) = 0$. This can be trivially verified.
   (a) $t < p(x_1, \cdots, x_{n-1}) \Rightarrow q(t, x_1, x_2, \cdots, x_{n-1}) = 0$. By definition of q.
   (b) $q(t, x_1, x_2, \cdots, x_{n-1}) = 0$. We already know that $p(x_1, \cdots, x_{n-1}) = 0$, then, by q's definition $t < 0$. Therefore, $t < p(x_1, \cdots, x_{n-1})$

2. $p(x_1, \cdots, x_{n-1}) > 0$.
   (a) $t < p(x_1, \cdots, x_{n-1})$. By the initial observation, we know that $\exists j$, such that,

$$p(x_1, \cdots, x_i, \cdots, x_{n-1}) = p(x_1, \cdots, x_i - j, \cdots, x_{n-1}) + jdt(x_i) \tag{7.14}$$

where $p(x_1, \cdots, x_i - j, \cdots, x_{n-1}) = 0$. Then $t < jdt(x_i)$, which implies $t - jdt(x_i) < 0$. Evaluating q in $(t - jdt(x_i)), x_1, \cdots, x_i - j, \cdots, x_{n-1})$, we find that

$$q(t - jdt(x_i), x_1, \cdots, x_i - j, \cdots, x_{n-1}) = 0 \tag{7.15}$$

Since, p is monotonic, $p(x_1, \cdots, x_i - (j-1), \cdots, x_{n-1}) > p(x_1, \cdots, x_i - j, \cdots, x_{n-1}) = 0$. Then,

$$q(t - jdt(x_i), x_1, \cdots, x_i - j, \cdots, x_{n-1}) = \tag{7.16}$$
$$q(t - (j-1)dt(x_i), x_1, \cdots, x_i - (j-1), \cdots, x_{n-1}) = 0 \tag{7.17}$$

Applying another $j - 1$ steps we obtain

$$q(t, x_1, \cdots, x_i, \cdots, x_{n-1},) = 0 \tag{7.18}$$

   (b) $q(t, x_1, x_2, \cdots, x_{n-1}) = 0$. We already know that $p(x_1, \cdots, x_{n-1}) > 0$. By definition,

$$q(t, x_1, \cdots, x_i, \cdots, x_{n-1}) = q(t - dt(x_i), x_1, \cdots, x_i - 1, \cdots x_{n-1}) \tag{7.19}$$

Recursively applying q's definition, we know that there exists j, such that,

$$q(t, x_1, \cdots, x_i, \cdots, x_{n-1}) = q(t - jdt(x_i), x_1, \cdots, x_i - j, \cdots, x_{n-1}) = 0 \tag{7.20}$$

and

$$p(x_1, \cdots, x_i, \cdots, x_{n-1}) = p(x_1, \cdots, x_i - j, \cdots, x_{n-1}) + jdt(x_i) \tag{7.21}$$

where $p(x_1, \cdots, x_i - j, \cdots, x_{n-1}) = 0$. By definition of q, $t - jdt(x_i)) < 0$. Then,

$$t - jdt(x_i)) < p(x_1, \cdots, x_i - j, \cdots, x_{n-1}) \tag{7.22}$$

Applying j steps of Eq. 7.6 will provide the desired result

$$t < p(x_1, \cdots, x_i, \cdots, x_{n-1}) \tag{7.23}$$

A similar theorem holds for a control expression that is strictly monotonic decreasing. For sake of space we avoid here its enunciation.

7.5.2 Obtaining a Generic Processor

After defining a set of parameters we need to build the description of a generic processor which, after replicating it and interconnecting its different instances, would produce the processor array to compute the space-time description. In figure 7.5, we present the general scheme for a behavioral description of a generic processor of the array. First we need to specify the input and output ports. They represent the inputs and outputs that are modeled in the behavioral architecture. They are also used to interconnect different instances of the generic processor. The second part is devoted to the behavioral description. We will use a process for each variable handled in the processor array. The set of processes are modeled to execute in concurrent fashion so that it accurately describes the behavior of variables. The code inside each process is modeled to be interpreted in sequential fashion. It will contain the necessary instructions to model predicates, to evaluate conditions and to respond to clock cycles.

```
ENTITY generic-processor IS
    PORT(
       input ports :  in . . .;
       output ports :  out . . .      );
END generic-processor;

ARCHITECTURE behavioral OF generic-processor
IS
    SIGNAL . . .: . . .;
BEGIN
    X: PROCESS( . . . )
       BEGIN
          . . .;
       END PROCESS;
    Y: PROCESS( . . . )
       BEGIN
          . . .;
       END PROCESS;
    Z: PROCESS( . . . )
       BEGIN
          . . .;
       END PROCESS;
END behavioral;
```

Figure 7.5: A typical behavioral VHDL description of a generic processor.

7.5.3 Processor Identification

In many situations is convenient for a processor to know its position within the entire array. For example, a processor located at the boundary of the array frequently needs to take some action to input or output values. Processor identification can be modeled in the following ways:

1. Internal register
2. Input port configuration
3. Generic port configuration

The first case is probably the best approach, closely related to the final implementation of the processor. However, that implies that each instance of the generic processor needs to be hardwired uniquely. From VHDL point of view, it is necessary to generate different instances, one per processor, which can be cumbersome.

Processor identification can be hardwired as a set of input ports. However, for synthesis purposes, the designated area for ports is very restricted and we need to avoid unnecessary ports.

A way to parameterize designs is through the use of generic parameters. They are used typically to transmit delays for simulation. We can use a generic parameter to indicate each processor identification. The syntax of constructs related to ports and generics is similar and it is shown in figure 7.6.

7.5.4 Port Configuration and Processor Interconnect

The main issue here is to determine the set of input and output ports for a generic processor. From the observations made about space-time representations, we know that each time we find an expression of the form

$$g_i(Y(\mathbf{z}-\delta),\dots) \tag{7.24}$$

```
ENTITY generic-processor IS
   GENERIC(
      id :  INTEGER( n-1 DOWNTO 0);
   );
   PORT(
      input ports :  in .  .  .;
      output ports :  out .  .  .
   );
END generic-processor;
```

Figure 7.6: A generic parameter to indicate processor identification.

we have to consider δ to determine if we need to add a register for delays, a communication between processor and, from that, we need to designate a port for that purpose. Each time we find a case

$$\mathbf{z} - \boldsymbol{\delta} = (t - \Delta t, x_1 - \Delta x_1, x_2 - \Delta x_2, \ldots, x_{n-1} - \Delta x_{n-1}) \tag{7.25}$$

where $\Delta_t \neq 0$ and $\exists \Delta_{x_i} \neq 0$, we need to add a communication link to the neighbor processor located in the direction and offset indicated by

$$\Delta_{\mathbf{x}} = (\Delta x_1, \Delta x_2, \ldots, \Delta x_{n-1}) \tag{7.26}$$

Therefore, it is necessary to add a port to read the value of variable Y. Similarly, an output port must be added for communicating Y-value yielded by this processor.

Let us suppose now that we have two dependency directions δ_1 and δ_2 which are the same except for the time component,

$$\begin{aligned} \delta_1 &= (\Delta t_1, \Delta x_1, \Delta x_2, \ldots, \Delta x_{n-1}) && (7.27) \\ \delta_2 &= (\Delta t_2, \Delta x_1, \Delta x_2, \ldots, \Delta x_{n-1}) && (7.28) \end{aligned}$$

and they appear upon the same variable Y. Since we need the value produced by the same processor at two different times, t_1 and t_2, we add only one communication interconnect and, hence, only one port for communicating such a variable. We take as a convention that, whenever we find a communication link of the type in eq. 7.25, we transmit it as soon as possible without considering the value of Δt. From here, we can interpret that the value taken at the input port would represent the value of Y on the corresponding processor delayed by 1 clock cycle.

7.5.5 Delay and Communication Management

Delays represent the way in which all processors of the array synchronize. A delay is required any time a dependency of the form

$$\mathbf{z} - \boldsymbol{\delta} = (t - \Delta t, x_1, \ldots, x_{n-1}), \qquad \Delta_t \geq 1 \tag{7.29}$$

is found. A delay can also be seen as a communication requirement. When the delay is of the form

$$\mathbf{z} - \boldsymbol{\delta} = (t - \Delta t, x_1, \ldots, x_{n-1}) \tag{7.30}$$

the communication is onto the processor itself. Let us consider the set of dependencies of the form in Eq. 7.30 referring to the same variable, Y,

$$S_Y = \{\delta | \mathbf{z} - \boldsymbol{\delta} = (t - \Delta t, x_1, \ldots, x_{n-1}), \quad \Delta t \geq 1, \quad X(\ldots, \mathbf{z} - \boldsymbol{\delta}, \ldots)\} \tag{7.31}$$

This set indicates that each processor is using values of Y produced at previous times, $t - \Delta_{t_1}, t - \Delta_{t_2}, \cdots, t - \Delta_{t_s}$. We propose the use of $\Delta_{t_{max}} - 1$ registers to store intermediate values, where

$$\Delta t_{\mathsf{max}} = \max\{\Delta t_1, \Delta t_2, \ldots, \Delta t_s\} \tag{7.32}$$

Registers are modeled in VHDL as signals, they can be updated with each clock cycle as shown in Figure 7.7.

```
SIGNAL Y1, Y2, Y: . . .
BEGIN
     . . .
b:  BLOCK( clk = '1' AND NOT clk'STABLE )
      BEGIN
         . . .
         Y <= ...;
         Y2 <= Y1;
         Y1 <= Y;
      END BLOCK;
```

Figure 7.7: Use of temporal signals for handling delays.

For values incoming from different processor, i.e. when

$$\mathbf{z} - \delta = (t - \Delta t, x_1 - \Delta x_1, x_2 - \Delta x_2, \ldots, x_{n-1} - \Delta x_{n-1}), \qquad \Delta t > 0, \qquad \exists i : \Delta x_i \neq 0 \tag{7.33}$$

if they refer to the same variable, since we already know that there exists one input to get the value of the variable delayed in one clock cycle. We use the same strategy to store as many values of the same variable incoming from the same processor as we need.

7.5.6 Condition Evaluation

According to the general structure of a recurrence equation representing space-time dimensions and to the discussion presented in the control-signal generation section, predicates of the form $t < p(x_1, x_2, \ldots, x_{n-1})$, $t = p(x_1, x_2, \ldots, x_{n-1})$, and $t > p(x_1, x_2, \ldots, x_{n-1})$, can be substituted, in many cases, for predicates of the form $q(t, x_1, x_2, \ldots, x_{n-1}) = 0$, $q(t, x_1, x_2, \ldots, x_{n-1}) = 1$, and $q(t, x_1, x_2, \ldots, x_{n-1}) = 2$, respectively, where q is itself defined by a recurrence involving time-independent predicates.

We now focus on the term $t = 0$ and $t < 0$ of Eq. 7.8. The first one represents the initial time or reset. The second term indicates when the processor array is working. We will use a special signal, RST, to know when the array needs to be re-initiated. The general VHDL model used for the evaluation of a variable X is shown in figure 7.8.

7.5.7 Generating the Regular Processor Array

Once we have determined the VHDL model for a generic processor we can build the processor array by adequately replicating it and interconnecting its different instances. We need to create a processor per point in the space domain of the space-time representation. Domains are commonly parallelepiped domains, then we can use the iterative features of VHDL which are similar to loop constructs in a programming language. To establish interconnections we need to consider the different dependencies, $\delta_1, \delta_2, \ldots, \delta_d$ in the space-time description. In general, we take

$$\Delta\mathbf{x} = \{(x_{1_i}, x_{2_i}, \cdots, x_{n-1_i}) | \delta_i = (\Delta t_i, \Delta x_{1_i}, \Delta x_{2_i}, \ldots, \Delta x_{n-1_i}), \quad 1 \leq i \leq d\} \tag{7.34}$$

To choose the ports to interconnect we examine the variable in which δ_i appears. In figure 7.9, we present the structure of the VHDL model for a linear processor array. We assume that the whole array has one input and one output and each processor has an interconnection with its previous neighbor processor.

7.6 A Case Study: The Edit Distance

Let $T = (t_1, t_2, \cdots, t_i, \cdots, t_n)$ and $R = (r_1, r_2, \cdots, r_j, \cdots, r_m)$ be two strings to compare. We are interested in the cost of transforming string R to string T. We can apply successive comparisons

```
ARCHITECTURE behavioral OF generic-processor IS
    . . .
BEGIN
    X: PROCESS( . . . )
       BEGIN
          . . .;
          IF RST = '1' THEN
             -- initialization
          ELSE
             -- get values from input ports
             CASE expr =>
                WHEN C1 => g1(···);
                WHEN C2 => g2(···);
                   . . .
                WHEN Ck => gk(···);
             END CASE;
             -- update signals for delayed values
             -- send computed values to the output port
       END PROCESS;
          . . .;
END behavioral;
```

Figure 7.8: A typical process model to compute the value of a variable in a clock cycle.

between elements of R and elements of T. When a mismatch occurs, we must consider the possibility of replacing a character of T for one of R, of inserting a character of T, or deleting a character of R. This problem can be well defined by the following recurrence relation stating a dynamic programming problem:

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + d(t_i,r_j) \\ D(i-1,j) + K_a \\ D(i,j-1) + K_o \end{cases} \tag{7.35}$$

with the initial conditions:

$$\begin{aligned} D(0,0) &= 0 \\ D(i,0) &= D(i-1,0) + K_a && \text{for } 1 \leq i \leq n \\ D(0,j) &= D(0,j-1) + K_o && \text{for } 1 \leq j \leq m \end{aligned}$$

where $d(t_i,r_j)$ represents the cost of replacing r_j by t_i, K_a the cost of adding t_i, and K_o the cost of omitting r_j. Note that in this formulation, the costs of insertion and suppression are constant, independent from the specific characters. In a typical application, like spelling correction, this calculation has to be repeated a lot of times since the same test string must be compared to many reference strings (a full dictionary for example). Therefore the amount of computation can be very large and prohibits the use of a conventional computer.

It is easy to see that the space-time description of Eq. 7.35 is given by

$$D(t,x) = \begin{cases} t=0, x=0 \rightarrow & 0 \\ t>0, x=0 \rightarrow & D(t-1,x)+Ko \\ 1 \leq t \leq n, t=x \rightarrow & D(t-1,x-1)+Ka \\ t>x, 1 \leq x \leq m \rightarrow & \min\{D(t-2,x-1)+d(t_{t-x},r_x), \\ & D(t-1,x)+K_a, D(t-1,x-1)+K_o\} \end{cases} \tag{7.36}$$

According to the discussion about control-signal generation we can avoid the computation of predicates $t = x$ and $x < t$. It is clear that those predicates are monotonically increasing in the

```
ENTITY regular-array IS
    PORT( I1:  in .  .  .; O1:  out .  .  .  );
END regular-array;

ARCHITECTURE iterative OF regular-array IS
    COMPONENT Gen-P
       GENERIC( id:  INTEGER( n1 DOWNTO 0 );
       PORT( In1:  in ...; Ou1:  out ...  );
    END COMPONENT;
    FOR ALL: Gen-P USE ENTITY work.Generic-Processor;
    SIGNAL S: vector( 0 TO N-1 );
BEGIN
    C0:  Gen-P
       GENERIC MAP( 0 );
       PORT MAP( I1, S(0) );

    C1TON-2:  FOR i IN 1 TO n-2 GENERATE
       C: Gen-P
          GENERIC MAP( i );
          PORT MAP( S(i-1), S(i) );
    END GENERATE;

    Cn-1:  Gen-P
       GENERIC MAP( n-1 );
       PORT MAP( S(n-2), O1 );
END iterative;
```

Figure 7.9: VHDL model for a linear processor array using the iterative features of VHDL.

x-dimension and the corresponding time dependency is $dt(x) = 1$. Then, we can write

$$D(t,x) = \begin{cases} t=0, x=0 \rightarrow & 0 \\ t>0, x=0 \rightarrow & D(t-1,x)+Ko \\ q(t,x)=1 \rightarrow & D(t-1,x-1)+Ka \\ q(t,x)=2 \rightarrow & \min\{D(t-2,x-1)+d(s_{t-x},r_x), \\ & D(t-1,x)+K_a, D(t-1,x-1)+K_o\} \end{cases} \tag{7.37}$$

where

$$q(t,x) = \begin{cases} x=0 \rightarrow & \begin{cases} t=0 \rightarrow & 1 \\ t>0 \rightarrow & 2 \end{cases} \\ x>0 \rightarrow & q(x-1,t-1) \end{cases} \tag{7.38}$$

Observing equations 7.37 and 7.38 we see that we need to store the value of D in each processor indicated by dependency $(t-1,x)$. Also, we need the value of D from previous processor provided by dependencies $(t-1,x-1)$ and $(t-2,x-1)$. Furthermore, we need the value of q from the previous processor provided at time $t-1$. The port configuration of the generic processor can be seen in figure 7.10.

In Figures 7.12, 7.13, and 7.14, we show the VHDL modeling of variables R, D, and Q of the generic processor. They are modeled by defining a process per variable. The code inside each process is sequential and the general structure is similar. The first part is devoted to initializing the instance, when RST is on. The second part evaluates each condition according to the predicates produced by the control-signal generation step.

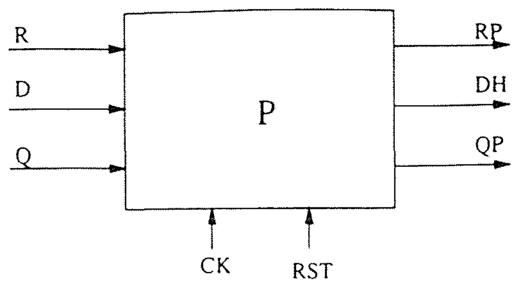


Figure 7.10: Port configuration of the generic processor for the edit distance problem.

```
ENTITY sm IS
    PORT(
       R : in BIT_VECTOR(3 DOWNTO 0);
       D : in BIT_VECTOR(3 DOWNTO 0);
       Q : in BIT
       RP : out BIT_VECTOR(3 DOWNTO 0);
       DH : out BIT_VECTOR(3 DOWNTO 0);
       QP : out BIT
       ck :  in BIT
       rst :  in BIT
    );
END sm0;
```

Figure 7.11: Port configuration of the generic processor for the edit distance problem.

7.7 Conclusions

We have discussed the main issues in generating behavioral VHDL models from space-time descriptions. Space-time descriptions are the final description of a systolic algorithm. Since they are intended to be implemented on hardware, a circuit must be obtained from them. Describing circuit designs can be done at various levels of abstraction. With the development of modern synthesizers, more often many circuits are described at behavioral level in which a circuit design is described by its input-output response. The development of general behavioral synthesizers is currently a matter of great research. See [32, 45] for a survey.

A behavioral VHDL model of a space-time description involves several issues. We discussed the most important ones. The control signal generation is of relevance since we must to save space in a circuit implementation. Several VHDL modeling features are used to obtain behavioral models of a space-time description. Iterative instantiation of components is used to obtain the regular processor array. Processor identification is achieved through generic parameters. Each variable is modeled through a separate and independent VHDL process. We discussed how to generate a generic processor for the array. We indicated how to determine the port configuration of it. Also, we gave a general structure for handling delays and for evaluating conditions of a variable.

Previous guidelines were determined by generating behavioral models for a set of dynamic programming algorithms. We showed the case of the edit distance problem. Those models were simulated to verify their behavior. Some of them were submitted to a behavioral synthesizer which successfully generated design at gate level. Although the designs obtained were effective and feasible, further work is required to reduce space of circuits and to speed the performance of the target design. Finally, given the restricted scope of space-time descriptions we think that it is possible to systematize the generation of design at a lower level such as the register-transfer level.

```
R: PROCESS( rst, ck )
   SIGNAL Ri:  REG_VECTOR(3 DOWNTO 0) REGISTER;
   BEGIN
      IF rst = '1' THEN
         Ri <= B"0000";
      ELSE            IF ck = '1' AND NOT ck'STABLE THEN
            Ri <= R;
         ENDIF;
      ENDIF;
      RP <= Ri;
   END PROCESS;
```

Figure 7.12: Process model of variable R of the generic processor for the edit distance problem.

```
D: PROCESS( rst, ck )
   SIGNAL D1, D2:  REG_VECTOR(3 DOWNTO 0) REGISTER;
   SIGNAL Di, Di1:  REG_VECTOR(3 DOWNTO 0) REGISTER;
   BEGIN
      IF rst = '1' THEN
         D1 <= B"0000";
         D2 <= B"0000";
      ELSE            IF ck = '1' AND NOT ck'STABLE THEN
         D2 <= D1; D1 <= D; Di1 <= Di;
         CASE ID =>
            WHEN '0' => Di <= Di1 + Ko;
            WHEN OTHERS
            CASE Q =>
               WHEN '0' => Di <= D1 + Ka;
               WHEN '1' =>
                  Di <= D2+mem(Ri);
                  Di <= MINSEL( Di, Di1+Ka );
                  Di <= MINSEL( Di, D1+Ko );
            END CASE;
         END CASE;
         ENDIF;
         DP <= Di;
      ENDIF;
   END PROCESS;
```

Figure 7.13: Process model of variable D of the generic processor for the edit distance problem.

```
Q: PROCESS( rst, ck )
   SIGNAL Q1:  BIT;
   SIGNAL Qi:  BIT;
   BEGIN
      IF rst = '1' THEN
         Q1 <= '0';
         Qi <= '0';
      ELSE            IF ck = '1' AND NOT ck'STABLE THEN
         Q1 <= Q;
         CASE ID =>
            WHEN '0' => Qi <= '1'
            WHEN OTHERS => Q1;
         END CASE;
         ENDIF;
         QP <= Qi;
      ENDIF;
   END PROCESS;
```

Figure 7.14: Process model of variable Q of the generic processor for the edit distance problem.

Chapter 8

Conclusions

8.1 Review of Goals

In this thesis we have presented an approach to synthesizing data-parallel algorithms of dynamic programming problems at the hardware description level.

Dynamic programming algorithms are particularly suitable for FPGA implementation, given that they require a very simple arithmetic and logical operations. Furthermore, the great number of dependencies that they exhibit require a fast communication medium for subproblem solutions. In that case, hardware implementation of DP algorithms is of great benefit to achieve good performance. Finally, the regularity shown in DP algorithms, both in computation and communication, is adequate to take advantage of the regularity in the FPGA's architecture.

We studied the characteristics of DP algorithms. In Chapter 2, we reviewed the general formulation of a DP algorithm and we showed two major classifications of DP algorithms. We also presented a set of DP algorithms to understand the nature of every class.

In Chapter 3, we presented general strategies to design data-parallel algorithms for DP problems. We determined that the important characteristics of a DP problem for data-parallel implementations are the size of the DP table, the seriability of the DP recurrence formulation, and the number of subproblem dependencies. We presented a naive approach to implement data-parallel dynamic programming in C* in which we designated a virtual processor for each subproblem of the DP table. Then, we showed a more practical approach to implement data-parallel algorithms by grouping several subproblems to be computed in a single processor. For uniform dependencies this approach is effective and it can be easily applied. For non-uniform dependencies, it is necessary to consider the dependency direction vectors. Finally, we further elaborate our data-parallel approach for DP problems having non-fixed number of dependencies. Combining non-uniform block decomposition, overlapping computation and communication, and achieving a balanced computational load for each processor, we obtained good performances in two real multicomputers (Meiko CS2 and CM-5).

In Chapter 4 we introduced a general terminology to describe recurrence equations from the parallel algorithm description perspective. We should say that the description of recurrence equation from the perspective found in the literature is very simple and brief. Then, the description presented in Chapter 4 is the most complete approach we can find in the literature of recurrence equations considering the parallel algorithm approach. We defined precisely the concepts of domains, identifiers, and dependency relation. We introduced both the general case of recurrence equations as well as the special cases.

We discussed in Chapter 5 a dataflow analysis technique for extracting the functional information from affine C* segment programs. The affine C* program fragments (ACPF) are descriptive enough to represent the class of data-parallel programs for dynamic programming algorithms and some other classes including linear algebra algorithms. The techniques shown in that Chapter capture the functional description of an ACPF producing recurrence equations describing it. The dataflow analysis technique is realized as an extension of techniques developed to analyze sequential programs. Given that the semantics of C* is rather different from that of a sequential programming language, we redefined the concepts of sequencing predicate and dataflow dependency relation. From the dataflow dependency relation of each parallel statement in an ACPF, we obtain the recurrence equations. We showed some examples for some kinds of ACPFs; those found in dynamic programming.

In Chapter 6, we examined the process of uniformization of recurrence equations. We examined the most frequent mathematical transformations used in that process. We introduced a representation for polyhedral domains. We also discussed the uniformization process in three examples: the convolution problem, the all-pairs shortest path problem and the gap problem. To the best of our knowledge no previous uniform representation for the gap problem has been found. We also introduced a library of functions to perform the uniformization process. Finally, we outlined the design of a tool for manipulating recurrence equations which is of utility in the uniformization process.

Obtaining space-time equations from uniform recurrence equations is perhaps the most studied topic in the systolic processing field. Very mature techniques exists to realize such transformation. The work done separately by Quinton [67], Rajopadhye [70], and Rao [71] are good examples of a systematic study for that topic. For such reason, we do not spend too much time in examining the problem.

Finally, in Chapter 7 we discussed to way to obtain VHDL descriptions from space-time representations. Since we can describe a circuit at various abstraction levels, we chose the highest one (behavioral) to describe our circuits. We discussed the main issues in obtaining such descriptions. We showed how to face the control signal generation, the delay and communication management, and the processor configuration and interconnection. Several designs for dynamic programming algorithm were produced following the approach presented in that chapter. They were effectively simulated through the Altera development system.

In summary, we can say that in this thesis we provide a clear methodology to synthesize algorithms from the data-parallel level to the hardware description level. Since our primary intention was to implement the algorithms in FPGAs, we restricted ourselves to DP algorithms because of the reasons clearly stated in chapters 1 and 2. However, we think that this methodology can be applied to a wider class of algorithms including, for example, linear algebra algorithms.

8.2 Open Problems and Future Work

In every investigation, there still exist some problems whose solution cannot be explicitly treated. Here we want to present three that our research did not address. They represent big questions for the parallel processing community:

1. If we relax timing functions, for example $t(z) = P(z)$, can we expand the class of recurrence equations for which a valid schedule and mapping exists? In other words, can the class of solvable recurrence equations be wider if we do not consider linear timing functions?. Feautrier (1992) proposed multidimensional timing functions [31] as a methodology.

2. Is there a class of recurrence equations for which a uniformization algorithm can be depicted? It is already known that problems with affine dependencies, $Az+b$, can be effectively converted to a uniform shape, is it possible to expand that class?

3. Usually mapping is the last step in the process of systolic synthesis. What would happen to the process if the first step that we do is mapping and then we look for timing functions?

In addition, we would like to present some extensions to the research we did that can be considered in the future.

1. First, it would be convenient to take the VHDL behavioral descriptions to FPGA implementation. That work would allow to exploration of performance considerations which were not considered in this research.

2. For developing efficient circuits, it would be more convenient to transform space-time descriptions to a lower abstraction level. Currently, work is being performed to transform space-time descriptions to register transfer level.

3. Many different space-time descriptions can be derived from the same set of uniform recurrence equations. There exist different optimization criteria for space-time descriptions, for example speed or circuit area. The tool shown here does not consider these factors and it would be good to incorporate such facilites.

4. The dataflow analysis of parallel programs can be extended by considering conditional loops (like the while-loop). Collard [23] developed an algorithm to analyze sequential while-loops. Our dataflow analysis technique can be modified to analyze while-loops of data-parallel statements.

5. The non-homogeneous block decomposition technique for implementing data-parallel programs can be applied to other kinds of problems showing a similar behavior to the one of the gap problem. Research work is required to find problems having non-fixed dependencies.

6. Finally, we think that the methodology presented here can be applied to algorithms other than dynamic programming ones. Some work needs to be done to show the feasibility of the methodology for other kind of algorithms.

8.3 Final Considerations

The realization of a research involves many steps and considers many aspects of the topic. The approach presented here was started from a general objective and this was divided into several subgoals to present a coherent solution. Our primary concern at the beginning was the implementaticn of algorithms for FPGAs. Soon, we became awar of the lack of techniques for this purpose starting from high-level specifications. For that reason, we concentrate on algorithms implemented in a high-level programming language. C* has been implemented with some degree of success in both multiprocessors and multicomputers. This research has tried to discover if C* can be applied to FPGA custom computing. In this thesis, the answer to that question has been partially answered. Here, we presented a methodology to transform a DP algorithm to hardware description level but, clearly, some steps need to be further investigated. In addition, some problems, like speed, area, and partitioning, might appear in the course of a specific FPGA implementation.

At every stage of the methodology proposed here we tried to make, whenever was possible, some contributions. The methodology along with the set of contributions discussed at every step represent the main contribution of this dissertation.

Although FPGA technology is being used more frequently for custom computing, at this moment reconfigurable architectures are rare and they have being developed mainly at academic laboratories. The future of custom computing is promising; the current examples of real reconfigurable architectures show its feasibility. Two main drawback have to be considered in order that reconfigurable architectures become commonly used. The first one is due to technological restrictions. Although the FPGA density has been increased recently, it is still not enough for many algorithmic applications. However, it is very probable that density will increase as technology evolves. The second drawback is concerned with the lack of techniques for implementing algorithms in circuits. Very few people would be interested in implementing algorithms by designing circuits. Clearly, compilation techniques are demanded to perform such task in an automatic way. We think that this research contributes in pursuing that goal.

Appendix A

Acronyms

ACPF	Affine C* Program Fragment
ARE	Affine Recurrence Equation
CLB	Configurable Logic Block
DP	Dynamic Programming
FPGA	Field Programmable Gate Array
OMP	Optimal Matrix Parenthesization
RE	Recurrence Equation
RTL	Register Transfer Level
SRE	System of Recurrence Equations
SURE	System of Uniform Recurrence Equations
URE	Uniform Recurrence Equation
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

Bibliography

[1] Corporation Actel. *ACT Family Field Programmable Gate Arrays Data Book*. Actel Corporation, Sunnyvale, CA, 1990.

[2] L. Agarwal, M. Wazlowski, and S. Ghosh. An asynchronous approach to efficient execution of programs on adaptive architectures utilizing fpgas. In *Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 101–110, 1994.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[4] Ltd. Algotronix. *Configurable Array Logic User Manual*. Algotronix, Ltd., Edinburgh, UK, 1991.

[5] Corporation Altera. *ALTERA Data Book 1987*. Altera Corporation, Santa Clara, CA, 1987.

[6] R. Andonov and S. Rajopadhye. Optimal tiling. Technical Report Publication Interne No. 792, IRISA, January 1994.

[7] Rumen Andonov and Sanjay Rajopadhye. Knapsack on VLSI: from algorithm to optimal circuit. *IEEE Transactions on Parallel and Distributed Systems*, 8(6), June 1997.

[8] P. M. Athanas and H.F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, pages 11–18, March 1993.

[9] D. H. Ballard and C. M. Brown. *Computer Vision*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

[10] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[11] M. Barnett. A systolizing compiler. Technical Report TR-92-13, University of Texas at Austin, May 1992.

[12] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[13] R. Bellman and S. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, Princeton, NJ, 1962.

[14] P. Bertin and H. Touati. PAM programming environments: Practice and experience,. In *Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 133–138, 1994.

[15] Phillip Gnassi Bradford. *Parallel Dynamic Programming*. PhD thesis, Indiana University, December 1994.

[16] J. Bu and E. F. Deprettere. Converting sequential iterative algorithms to recurrence equations for automatic design of systolic arrays. In *IEEE International Conference on Acoustic, Speech and Signal Processing*, pages 2025–2028, May 1988.

[17] H. D. Chen and K. S. Fu. Algorithm partition and parallel recognition of general context-free languages using fixed-sized vlsi architecture. In Ohio State University, editor, *Proceeding of the Midwest VLSI Workshop*, Jan 1985.

[18] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, pages 461–491, 1986.

[19] M. C. Chen. A parallel language and its compilation to multiprocessor machines for VLSI. In *Principles of Programming Languages*. ACM, 1986.

[20] M. C. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *Journal of Supercomputing*, 2:171–207, 1988.

[21] Y. T. Chiang and K. S. Fu. Parallel parsing algorithms and VLSI implementations for syntactic pattern recognition. *IEEE Transaccions on Pattern Analysis and Machine Intelligence*, PAMI-6, 1984.

[22] M. J. Clark and C. R. Dyer. Curve detection in VLSI. In K. S. Fu, editor, *VLSI for Pattern Recognition and Image Processing*. Springer-Verlag, Berlin, 1984.

[23] J.-F. Collard. Space-time transformation of while-loops using speculative execution. In IEEE, editor, *Proc. of the 1994 Scalable High Performance Computing Conference*, pages 429–436, May 1994.

[24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, New York, NJ, 1990.

[25] A DeHon. DPGA-coupled microprocessors: Commodity ics for the early 21st century. In *Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, 1994.

[26] David Epstein, Zvi Galil, Giancarlo Raffaele, and Giuseppe F. Italiano. Sparse dynamic programming I: Linear cost function. *Journal of the Associate Computing Machinery*, 39(3):519–545, July 1992.

[27] David Epstein, Zvi Galil, Giancarlo Raffaele, and Giuseppe F. Italiano. Sparse dynamic programming II: Convex and concave cost functions. *Journal of the Associate Computing Machinery*, 39(3):546–567, July 1992.

[28] B. Fagin and J. G. Watt. FPGA and rapid prototyping technology use in a special purpose computer for molecular genetics. In *Proceedings of the IEEE International Conference on Computer Design*, pages 496–501, 1992.

[29] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[30] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, October 1992.

[31] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[32] D. Gajski, N. Dutt, A. Wu, and Y. Lin. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, MA, 1992.

[33] Z. Galil and K. Park. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989.

[34] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than $O(1)$ dependency. *Journal of Parallel and Distributed Computing*, 21:213–222, 1994.

[35] Zvi Galil and Kungsoo Park. Dynamic programming with convexity, concavity, and sparsity. *Theoretical Computer Science*, 92:49–76, 1992.

[36] M. Gokhale, W. Holmes, A. Kosper, S. Lucas, R. Minnich, and D. Sweely. Building and using highly parallel programmable logic array. *IEEE Computer*, pages 81–89, January 1991.

[37] M. Gokhale and R. Minnich. FPGA computing in a data parallel C. In *Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 94–101, 1993.

[38] J. Greene, E. Hamdy, and S. Beal. Antifuse field programmable gate arrays. *Proceeding of the IEEE*, 81(7):1993, July 1042-1055.

[39] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation of combinatorial algorithms. In California Institute of Technology, editor, *Proceedings of Conference on Very Large Scale Integration*, pages 509–525, 1979.

[40] P. J. Hatcher, M. J. Quinn, A. J. Lapadula, B. K. Seevers, R. J. Anderson, and R. R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[41] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* The MIT Press, Cambridge, MA, 1991.

[42] D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM Journal of Computing*, 16(4):628–638, 1987.

[43] Hu. *Integer Programming and Network Flows.* Addisson-Wesley, Reading, MA, 1969.

[44] S. S. Huang, H. Liu, and V. Viswanathan. Parallel dynamic programming. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):326–328, 1994.

[45] Ahmed Amine Jerraya, Hong Ding, Polen Kission, and Maher Rahmouni. *Behavioral Synthesis and Component Reuse with VHDL.* Kluwer Academic Publishers, Norwell, MA, 1997.

[46] G. Karipys and V. Kumar. Efficient parallel formulations for some dynamic programming algorithms. Technical Report TR 92-59, Comp. Sc. Department, University of Minnesota, October 1992.

[47] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[48] Randy H. Katz. *Contemporary Logic Design.* Addisson-Wesley, Reading, Mass., 1994.

[49] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamming Cummings Pub. Co., Redwood City, CA, 1994.

[50] S. Y. Kung. *VLSI Array Processors.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[51] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[52] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3(3):173–182, September 1991.

[53] G. Li and B. W. Wah. Systolic processing for dynamic programming problems. In *Proc. of International Conference on Parallel Processing*, pages 434–441, 1985.

[54] H. H. Liu and Fu. K. S. VLSI arrays for minimum distance classification. In K. S. Fu, editor, *VLSI for Pattern Recognition and Image Processing.* Springer-Verlag, Berlin, 1984.

[55] D. P. Lopresti. P-NAC: a systolic array for comparing nucleic acid sequences. *IEEE Computer*, 20(7):98–99, February 1987.

[56] D. E. Maydan, S. P. Amarasinghe, and M. L. Lam. Array data-flow analysis and its use in array privatization. In *Proceedings of ACM Conference on Principles of Programming Languages*, pages 2–15, January 1993.

[57] D. E. Maydan, J. L. Hennesy, and M. S. Lam. Effectiveness data dependence analysis. In *Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures*, 1992.

[58] V. Mazlov. Lazy array data-flow dependence analysis. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 311–325, January 1994.

[59] S. Mazor. *A Guide to VHDL.* Kluwer, San Mateo, CA., 1993.

[60] D. I. Moldovan. ADVIS: A software package for the design of systolic arrays. *IEEE Transactions on Computer-Aided Design*, CAD-6(1):33–40, 1987.

[61] R. K. Moore. A dynamic programming algorithm for the distance between two areas. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, PAMI-1, 1979.

[62] C. S. Myers and L. R. Rabiner. A level building dynamic time warping algorithm for connected word recognition. *IEEE Transaction on Acoustics, Speech and Signal Recognition*, ASSP-29, 1981.

[63] John V. Oldfield and Richard C. Dorf. *Field Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems.* John Wiley & Sons, Inc., New York, NY, 1995.

[64] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

[65] W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. Technical Report CS-TR-3372, Comp. Sc. Department, University of Maryland, November 1994.

[66] Michael J. Quinn and Philip J. Hatcher. On the utility of communication-computation overlap in data-parallel programs. *International Journal on Parallel and Distributed Computing*, 33(2):197–204, March 1996.

[67] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. of the 11th Annual Symposium on Computer Architecture*, pages 208–214, 1984.

[68] P. Quinton and I. Robert. *Systolic Algorithms and Architectures.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[69] L. R. Rabiner, A. E. Rosenberg, and S. E. Levinson. Considerations in dynamic time warping algorithms for discrete word recognition. *IEEE Transaction on Acoustics, Speech and Signal Recognition*, ASSP-26, 1978.

[70] S. V. Rajopadhye and R. M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14:163–189, 1990.

[71] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proceeding of the IEEE*, 76(3):259–269, 1988.

[72] Hudson B. Ribas. *Automatic Generation of Systolic Programs From Nested Loops.* PhD thesis, Carnegie Mellon University, June 1990. Published as CMU-CS-90-143.

[73] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field programmable gate arrays. *Proceeding of the IEEE*, 81(7):1013–1029, July 1993.

[74] W. Rytter. Efficient parallel computations for dynamic programming. *Theoretical Computer Science*, 59:297–307, 1988.

[75] H. Sakoe and S. Chiba. Dynamic programming optimization for spoken word recognition. *IEEE Transaction on Acoustics, Speech and Signal Recognition*, ASSP-26:43–49, 1978.

[76] David Sankoff and Joseph B. Kruskal, editors. *Time warps, string edits, and macromolecules: The theory and practics of sequence comparison.* Addison-Wesley, Reading, MA, 1983.

[77] M. Sniedovich. *Dynamic Programming.* Marcel Dekker, Inc., New York, NY, 1992.

[78] S. Triemberger. A reprogrammable gate array and applications. *Proceeding of the IEEE*, 81(7):1993, July 1030-1056.

[79] N. H. E. Weste, D. H. Burr, and B. D Ackland. A systolic processing element for speech recognition. In *IEEE International Solid State Circuit Conference*, pages 274–275, 1982.

[80] D. Wilde. A library for doing polyhedral operations. Technical Report TI-785, IRISA, December 1993.

[81] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Readwood City, CA, 1996.

[82] Inc. Xilinx. *The Programmmable Gate Array Data Book.* Xilinix Inc., San Jose, Ca, 1992.