

1992

Multiplexed pipelining : a cost effective loop transformation technique

Satish Pai
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds



Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Pai, Satish, "Multiplexed pipelining : a cost effective loop transformation technique" (1992). *Dissertations and Theses*. Paper 4425.

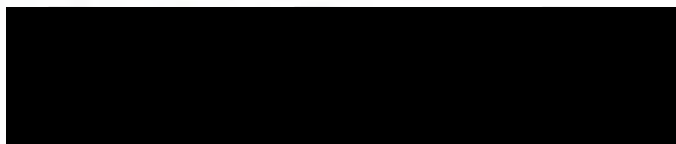
<https://doi.org/10.15760/etd.6303>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

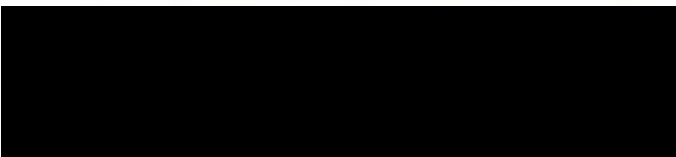
AN ABSTRACT OF THE THESIS OF Satish Pai for the Master of Science in Electrical Engineering presented May 22, 1992.

Title: Multiplexed Pipelining: A Cost Effective
Loop Transformation Technique

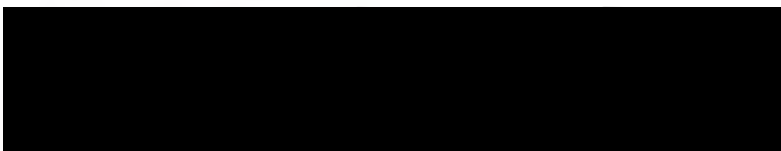
APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:



Michael A. Driscoll, Chair



W. Robert Daasch



Andrew Fraser

Parallel processing has gained increasing importance over the last few years. A key aim of parallel processing is to improve the execution times of scientific programs by mapping them to many processors. Loops form an important part of most computational programs and must be processed efficiently to get superior performance in terms of execution times. Important examples of such programs include graphics algorithms, matrix operations (which are used in signal processing and image processing applications), particle simulation, and other scientific applications. Pipelining uses overlapped parallelism to efficiently reduce execution time.

Loop transformations exploit the potential parallelism within loops by rearranging loop components to reduce execution time. Loop transformations can also be viewed as a solution to a mapping problem in which the computation space associated with the loops is mapped to processor space and time.

A new loop transformation method called multiplexed pipelining (MUP) is presented in this thesis. The MUP method considers the cost of operation and makes efficient use of a limited number of processors by using pipelining for mapping the loop computation space. The MUP method considers partitioning at a fine grain level including parallelism within individual statements. Execution models for the MUP method are developed. Other mapping methods are considered and execution models for them are developed. The models are used to evaluate MUP relative to the other methods in terms of cost, execution time and flexibility.

Our analysis shows that for different loop size parameters MUP is a suitable choice over other mapping methods. The choice for MUP can be made at compile time using the execution models. These execution models make use of static program characteristics such as the loop size parameters. Hence, for a given loop, MUP can be selected as the best mapping before the program executes.

The execution models for MUP are validated by empirical measurements of program loops. These measurements also show a speedup of as much as 4.9 on 6 processors for a small loop iterated 1600 times.

MULTIPLEXED PIPELINING: A COST EFFECTIVE
LOOP TRANSFORMATION TECHNIQUE

by

SATISH PAI

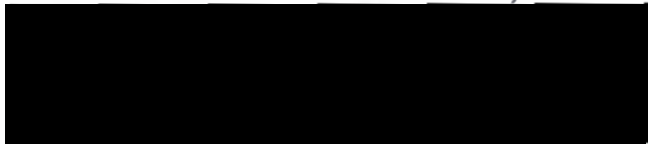
A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL ENGINEERING

Portland State University
1992

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Satish Pai
presented May 22, 1992.



Michael A. Driscoll, Chair

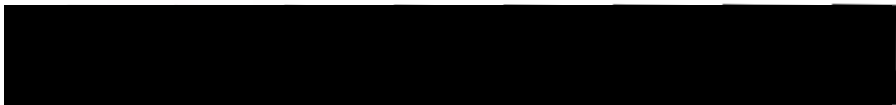


W. Robert Daasch



Andrew Fraser

APPROVED:



Rolf Schaumann, Chair, Department of Electrical Engineering



C. William Savery, Vice Provost for Graduate Studies and Research

TABLE OF CONTENTS

	PAGE
LIST OF TABLES	v
LIST OF FIGURES	vi
LIST OF SYMBOLS	vii
CHAPTER	
I	INTRODUCTION. 1
	I.1 Problem statement. 1
	I.2 Thesis objective. 1
	I.3 Thesis overview. 2
II	LOOP TRANSFORMATIONS. 3
	II.1 Structure of program loops. 3
	II.2 Loop transformation-definition. 4
	II.3 Pipelining-An example mapping. 7
	II.4 Computation space for our research. 11
	II.5 Exploration of important mapping methods. 13
	II.6 Motivation for our research. 17
III	PERFORMANCE EVALUATION FOR MULTIPLEXED PIPELINE METHOD. 19
	III.1 Multiplexed pipeline mapping. 19
	III.2 Exploring the solution space. 20
	III.3 Comparison of mapping methods. 30

IV	EXPERIMENTAL RESULTS AND ANALYSIS.	37
	IV.1 Types of experiments.	37
	IV.2 Graph model for experiments.	37
	IV.3 The experimental setup.	38
	IV.4 Measurement of the stage time.	39
	IV.5 Execution time experiments.	40
	IV.6 Speedup measurement.	46
	IV.7 Experimental result analysis.	49
V	SUMMARY AND CONCLUSIONS.	51
	REFERENCES.	53

LIST OF TABLES

TABLE		PAGE
I	Lower Execution Time between MNP and MUP Method for $E > K$.	33
II	Lower Execution Time between MNP and MUP Method for $E = K$.	33
III	Lower Execution Time between MNP and MUP Method for $E < K$.	33
IV	Calculation of Stage Time.	40
V	Execution Times for 400 iterations	41
VI	Execution Times for 800 iterations	41
VII	Execution Times for 1200 iterations.	42
VIII	Execution Times for 1600 iterations.	42
IX	Uniprocessor Execution Time(seconds).	47
X	Speedup results.	47

LIST OF FIGURES

FIGURE		PAGE
1.	Block Level Structure of a Loop code.	5
2.	Basic structure of a pipeline processor.	8
3.	A Simple pipeline operation.	8
4.	Computation space for pipelined operation.	9
5.	Computation Space for the loop structure.	12
6.	Multiplexed Pipeline Mapping.	19
7.	First case $K = S = E$	22
8.	General case LCF subspace.	28
9.	General case LCE subspace.	28
10.	Graph Model for Experiments.	38
11.	Execution Time For iterations 400 and 800.	44
12.	Execution Time For iterations 1200 and 1600.	45
13.	For $\Delta = 0, 1, 2, 3, 4$ and 5	46
14.	Pipelined Speedup for iterations 400 and 800.	48
15.	Pipelined Speedup for iterations 1200 and 1600.	49

LIST OF SYMBOLS

N	Total number of iterations
S	Total number of loop paths
E	Total number of sub steps in a path
K	Total number of processors used
Δ	Pipeline delay
Π	Mapping function for Time
P	Mapping function for processor space
T_{ex}	Total execution time
T_u	Number of units required for N iterations
T_m	Measured execution time
T_{pred}	Predicted execution time
T_s	Pipeline stage time
LC	Loop control statement
TH	Threads(processing statements)
A	Entry node in a path
ORD	Orderings of threads
ord	Orderings of paths(including loop control statement)

CHAPTER I

INTRODUCTION

I.1 PROBLEM STATEMENT

Parallel processing has become a necessary means for improving the execution times of scientific programs. The key task in parallel processing is mapping a sequential program to different processors for simultaneous execution. Efficient mapping of these programs requires the exploitation of potential parallelism within the program code.

Loops play an important role in most compute intensive scientific programs and efficient parallel execution of these programs requires a methodology for the efficient partitioning of programs and the scheduling of the program components within the partitions,[1]. Loop transformations and pipelining are often proposed as a way to automatically (or manually) improve the performance of scientific programs on high performance computer system. Pipelining improves performance by overlapping the execution of several different instructions. If there are no interactions between the instructions, several instructions can be in different stages of execution simultaneously. Pipelining also makes very efficient use of limited hardware resources. It is important, therefore, to develop pipeline -oriented loop transformations.

I.2 THESIS OBJECTIVE

The goal of our research is to develop a cost effective loop transformation method which improves the execution performance of program loops. The use of pipelining is considered to get the benefits of overlapped parallelism. The proposed method, called

multiplexed pipelining (MUP) is presented as a solution to the problem of mapping the computation space defined by a loop to processors and time. Fine grain partitioning of the loop is used to expose more parallelism. MUP considers the cost of the solution and makes efficient use of a limited number of processors. MUP is compared to other mapping methods for performance evaluation.

The evaluation process begins with the development of models for execution time and cost for each method. The models are used to evaluate the merits of each method. The models for MUP are validated via several experiments. Our analysis shows that for certain cost constraints, MUP outperforms the other methods.

I.3 THESIS OVERVIEW

Chapter II defines the nature of program loops. It describes the space-time representation of a computational process. Simple pipelining is used to illustrate mapping problem and the space-time notation. The computation space for our research is defined. Several common mapping methods are introduced and modeled using the space-time notation.

Chapter III describes the MUP method in detail. Execution time models are developed and are used to identify key factors determining performance. Finally MUP is compared with the mapping methods introduced in Chapter II.

Chapter IV discusses the design of experiments to validate the models for MUP's performance and presents results of the experiments. The results confirm the models. Finally Chapter V presents the summary and conclusions and discusses MUP's importance as compared with other mapping methods.

CHAPTER II

LOOP TRANSFORMATIONS

II.1 STRUCTURE OF PROGRAM LOOPS

The structure of a loop is illustrated by its pseudo code. Shown below is the pseudo code for a FORTRAN "DO" loop. The loop limits are arbitrary and chosen for this example.

```
DO 10 I = 1,100
```

```
    S1(I) => f(S11(I), S12(I),...,S1n(I));
```

```
    S2(I) => f(S21(I), S22(I),...,S2n(I));
```

```
    S3(I) => f(S31(I), S32(I),...,S3n(I));
```

```
    .
```

```
    .
```

```
    Sn(I) => f(Sn1(I), Sn2(I),...,Snn(I));
```

```
10 CONTINUE.
```

The pseudo code consists of two main components: loop control statement and processing statements. The loop control statement in the above example is represented by the DO statement and the CONTINUE statement. The processing statements are enclosed between the DO and CONTINUE statements. The DO statement specifies actions which define the initialization, comparison and increment functions for the loop. The term initialization refers to actions taken before the first loop iteration, in this case setting I to 1. The comparison function checks the loop limits at each iteration and either repeats or ter-

minates loop execution. In this case, I is compared with 100 and the loop is terminated when I is greater than 100. The increment function increments the iteration count for the next loop iteration, in this case I is incremented by 1.

The processing statements assign a value that is a function of its substeps to a variable. As an example, the following is a simple processing statement.

$$a = b + c - d$$

where the variable a is assigned the value of a function of b, c and d. The terms + and - represents the substeps in the statement.

Based on the above description, a graphical representation of the loop is given in Figure 1. The general structure of the loop includes four main blocks. The first block represents the initial portion of program before the loop body. The second block represents initialization for the loop operation. The third block represents the loop body which consists of loop control statement and processing statements. The fourth block represents the rest of the program.

II.2 LOOP TRANSFORMATION-DEFINITION

Loop transformations are a powerful collection of methods that improve the performance of a program by matching the program to architectures. Loop transformations effectively rearrange the loop components to enhance the execution speed on a specific multiprocessor system. Transformations of a program produce execution speed enhancement through the exploitation of inherent parallelism and by effectively exploiting memory hierarchies and interprocessor communication networks. In recent years many researchers including Wolfe[1], Kennedy[4], Kumar[2] and Banerjee[10] have looked at loop transformation methods as an important compilation tool for parallel processing systems.

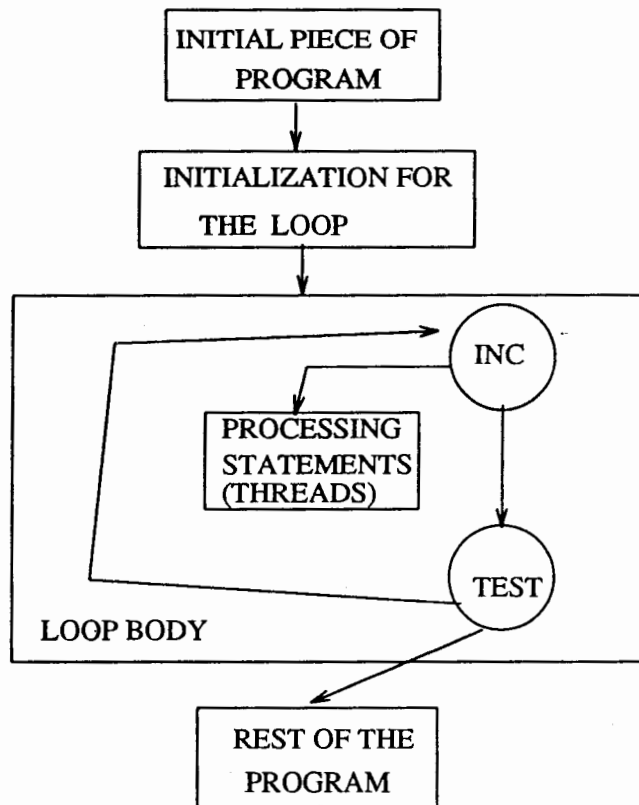


Figure 1.Block Level Structure of a Loop code.

II.2.1 Loop Transformation as a Mapping Problem

Loop transformation techniques can be defined as a process which maps the given computation space (or computation structure) into a processor space by partitioning and scheduling the partitions in an efficient way[11]. A computation space consists of nodes corresponding to computations and arcs showing the flow of data or the dependence between nodes.

A node in the computation space can represent the individual elements which are assigned to processing elements. Nodes of different grain sizes can be used. Miranker and Winkler[11] define two types of nodes for their computation. For uniprocessor execution, where a single processing element is used for computation, a node is the entire loop. For multiprocessor execution, nodes are individual iterations. Sheu and Tai[20]

define a node as a partition consisting of iterations which are independent of each other. Moldovan[12] defines a node as set of iterations for mapping the cyclic loop algorithms on VLSI arrays.

Various types of arc dependences can also be defined. Sheu and Tai[20] and Moldovan[12] consider the dependence existing between iterations and define the concept of a dependence vector. If a variable generated at iteration i is used later by iteration j , then the dependence relation for that variable is represented by a dependence vector from i to j . Shang and Fortes[16] define uniform data dependencies where the values of the dependence vectors are independent of iterations. Wolfe[9] considers the dependences existing between statements in a loop for the vectorization test. The term vectorization here refers to the conversion of code written for serial execution into code which uses the vector instructions of the target machine. Wolfe[9] also considers the possibility of dependences that can occur between substeps in a loop statement.

A processor space is a description of the processing elements available for the given computation. Two main types of processor spaces are of interest. The first one is a uniprocessor space and consists of a single processor. The second one is a multiprocessor space and consists of several processors. There exist various types of multiprocessor spaces based on the interconnection topology between the processors. Important examples include local area networks, n -dimensional hypercubes and systolic arrays.

Winkler and Miranker[11] consider both types of processor spaces. For the uniprocessor space they assume the coordinates of the given processor to be the origin(0,0,0). For the multiprocessor space, the coordinates of each processor is triple of integers. Sheu and Tai[20] use the n -dimensional hypercube topology as the processor space for their computation. The coordinates of each processor is a point in this n -dimensional space. Moldovan[12] uses VLSI systolic arrays as the processor space for the mapping of loop algorithms. In this case the position of each processing cell is

described by its two-dimensional Cartesian coordinates.

Time space can be combined with processor space to define a computation. A node in the given computation space is mapped to a specific coordinate in processor space and time, defining when and where the node executes.

Loop transformation techniques can be defined as the process of mapping the computation space to the given processor and time spaces. The physical process of computation can be interpreted in terms of physical space and time. Winkler and Miranker[11] define the above technique as the space-time representation of the computation space. Two distinct functions Π and P are defined by Moldovan[12]. Π represents the mapping from computation space to time space for the given computation process. P represents the corresponding mapping from computation space to processor space.

Several types of mapping methods have been considered. Moldovan[12] uses a linear mapping method to map the loops to VLSI systolic arrays. Sheu and Tai[20] use a hyperplane method which considers a linear mapping scheme to map the iterations to the n-dimensional hypercube. The widely used uniprocessor mapping is an example of a nonlinear mapping scheme. Other methods are intuitive methods like a separate processor assigned to every node of the computation space for each iteration.

II.3 PIPELINING-AN EXAMPLE MAPPING

Pipelining offers an economical way to realize parallelism in modern digital computers. The concept of a pipeline is similar to an assembly line in a manufacturing plant where the input task gets divided into a series of subtasks and each subtask can be executed by a specific stage that operates concurrently with other stages in the pipeline. All tasks are driven into the pipeline and executed in an overlapped fashion at the subtask level. Pipelined processing has led to tremendous performance improvement in modern computing systems.

II.3.1 Basic Structure

A basic linear pipeline is shown in Figure 2.

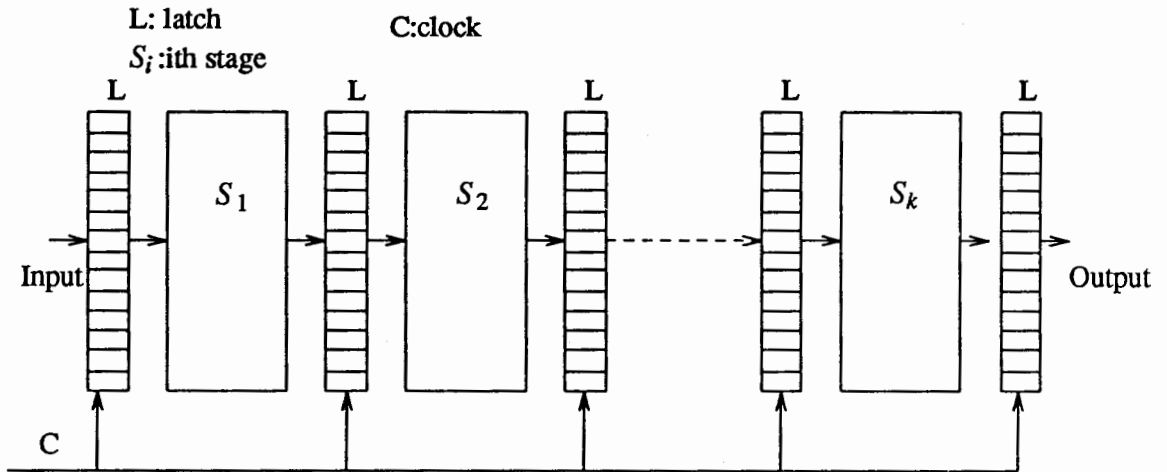


Figure 2. Basic structure of a pipeline processor.

The pipeline consists of a cascade of processing stages. The stages are separated by interface latches. The latches are fast registers used to hold the intermediate results. A common clock drives all stages simultaneously.

The speedup of a k -stage linear pipeline is defined to be the ratio of execution time on a non-pipelined processor to execution time on a pipelined processor.

Figure 3 gives the Gantt chart for the pipeline of Figure 2.

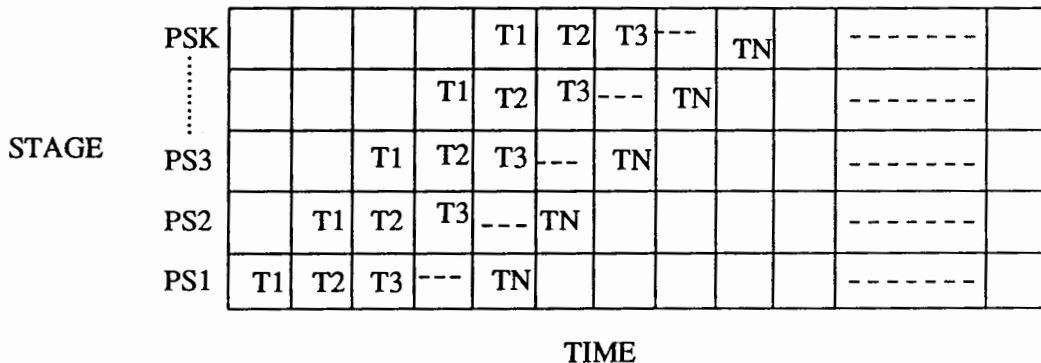


Figure 3. A simple pipeline operation.

T1, T2, etc represent the instructions in the pipeline. PS1, PS2, etc. represent the stages in the pipeline.

II.3.2 Pipeline Computation Space

Pipelined operation can be obtained via mapping process. For this purpose the computation space has to be defined. As mentioned above, the linear pipeline executes instructions which are made of subtasks. There is a precedence relation between adjacent subtasks in an instruction. The node of computation is an individual subtask. A computation space suitable for pipelining is shown in Figure 4.

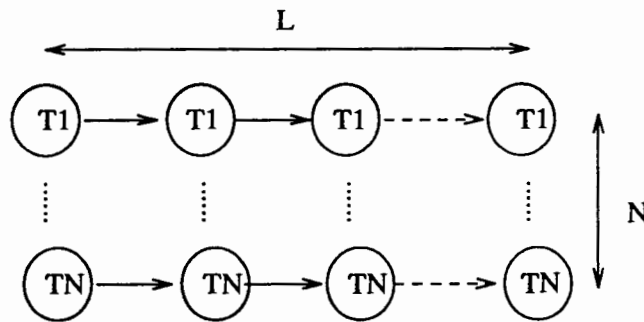


Figure 4.Computation space for pipelined operation.

Several variables define an instance of this computation space:

1. L represents maximum length (number of subtasks) of an instruction.
2. N represents total number of instructions.

An individual node is labeled (l,n) where $1 \leq l \leq L$ and $1 \leq n \leq N$.

II.3.3 Processor Space for Pipelined Computation

Two types of processor spaces can be considered for the pipeline computation space shown in Figure 4. They are:

1. Uniprocessor space: The pipeline computation space is mapped to a single processor.

2. Multiprocessor space: The pipeline computation space is mapped to several processors connected in a pipelined fashion. If K processors are used, an individual processor will be labeled p , where $1 \leq p \leq K$.

II.3.4 Uniprocessor Pipeline Mapping

This method employs a single processor for its operation. The processor mapping P , in this case is

$$P(l, n) = 1$$

The above equation means that entire computation takes place at the single processor. The time mapping function, Π , in this case is given by

$$\Pi(l, n) = (n - 1) * L + l \quad (1)$$

The equation for the mapping function Π consists of two terms. The first term, $(n - 1) * L$ represents the time taken to execute first $(n - 1)$ instructions and the second term, l represents the time mapping for the substeps of the n th instruction. The total execution time T_{ex} is the maximum value of Π and is given by

$$T_{ex} = \max_{l, n} \Pi = (N - 1) * L + L = N * L \quad (2)$$

This means that $N * L$ cycles are needed to complete the given computation using a single processor.

II.3.5 Multiprocessor Pipeline Mapping

In this case, the processor mapping P , is given by

$$P(l, n) = l$$

Since this mapping is pipelined, the overlap between the instructions must be considered. The time mapping function, Π , in this case is given by

$$\Pi (l, n) = (n - 1) [T_1] - (n - 1) [T_2] + [T_3]$$

where

T_1 represents the time to complete a single instruction;

T_2 is overlap between the instructions; and

T_3 is time mapping for the subtasks of the last instruction.

Substituting the values for each term, we get

$$\Pi (l, n) = (n - 1) [L] - (n - 1) [L - 1] + l \quad (3)$$

The physical interpretation of the above equation is as follows. It takes L cycles to complete a single instruction. The overlap between the successive instructions is $(L - 1)$ and a total of $(n - 1)$ overlapping instructions. The last term l represents the time mapping for the subtasks of the last instruction.

The total execution time T_{ex} is maximum value of Π and is given as

$$T_{ex} = \max_{l,n} \Pi = (N - 1) [L] - (N - 1) [L - 1] + L$$

Since $L = K$, Simplifying the above equation we get,

$$T_{ex} = K - 1 + N \quad (4)$$

As expected, this shows that $K - 1$ cycles are required to fill the pipeline and then N cycles are required to complete N tasks.

II.4 COMPUTATION SPACE FOR OUR RESEARCH

Figure 5 illustrates the computation space we will consider in this research. LC represents the loop control statement which includes the functions INC and TEST as shown in Figure 1. S1, S2, etc., represent the processing statements. The symbol i refers to the execution of the loop statement in the i th iteration. Circles represent the substeps

in the statement. There is a sequential dependence between these substeps. The term sequential dependence refers to the dependence from a node to its successor node. The nodes denoted by symbol A are the entry nodes for the statements. Entry nodes are initialized at the start of the execution.

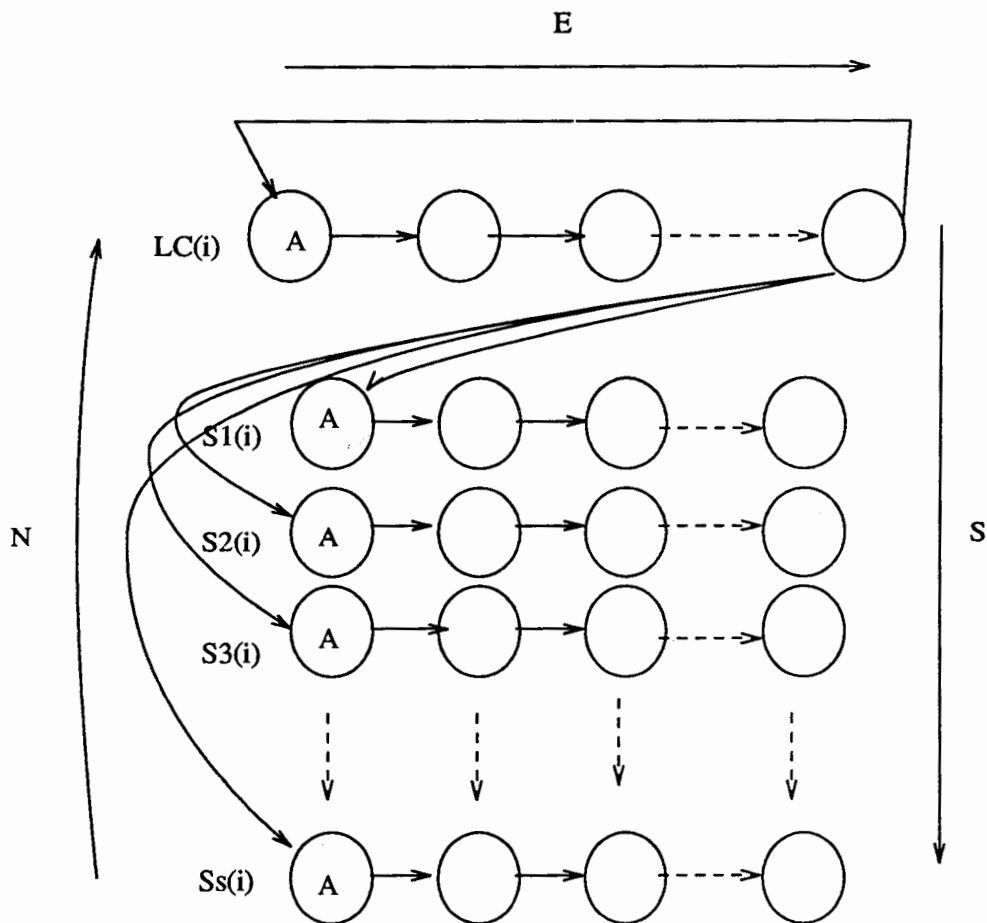


Figure 5.Computation Space for the loop structure.

Three variables define an instance of this computation space:

1. N : Total number of iterations, $(1 \leq i \leq N)$.
2. S : Total number of statements(includes loop control statement), $(1 \leq s \leq S)$.
3. E : Total number of substeps in a statement. All statements in this computation space assume equal length. $(E_s = E \text{ for all } S)$, $(1 \leq e \leq E)$.

The computation space shown in Figure 5 consists of two main loop components: loop control statement and processing statements. Both of these components can be realized as individual paths in the loop. These paths are made up of substeps with a sequential precedence relation existing between them.

The computation space considered in this research is similar to actual program loops. In fact, most loops can be transformed to our computation space, perhaps with a small loss of parallelism. This loss of parallelism stems from the pipelined nature of our computation space.

First, note that there is no dependence between statements in our computation space. For a loop having such dependence, all statements involved in the dependence can be merged into a single "super-statement". This may result in the loss of parallelism between the portions of the dependent statements that are independent. Second, an actual program statement may not require the linear ordering of our computation space. In other words, substeps of a statement may be executed in parallel. Such a statement can be mapped to our computation space by enforcing an arbitrary ordering on any substeps that could execute in parallel.

Finally, the statement lengths were assumed equal in our computation space. There exist a dependence arc from the loop control statement to other processing statements. This can cause delays in the pipelined execution, if the statements are of unequal length. Assuming that the statements are of equal length simplifies the analysis. A short statement in an actual loop can be padded with no ops to match our assumption. These no ops are identical to the delays a short statement would cause.

II.5 EXPLORATION OF IMPORTANT MAPPING METHODS

The mapping problem consists of assigning our computation space to the given processor space and time. Given this concept, many mappings can be suggested. We

have tried to look at a few important ones categorized in terms of processor space and degree of pipelining used.

II.5.1 Uniprocessor Method (UM)

This method employs a single processor for the entire computation. There is no parallelism or pipelining involved. This is a widely used mapping method and can be used in the cases where execution time is not a main criterion. This method is used in commercially available uniprocessor computers. For this case the processor mapping P is

$$P(i, s, e) = 1$$

A typical mapping function, Π , for this case is given by

$$\Pi(i, s, e) = (i - 1) * S * E + (s - 1) * E + e \quad (5)$$

The above equation represents the time mapping for the uniprocessor computation. The equation is split into three terms. The first term, $(i - 1) * S * E$ represents the time required for all the iterations before the i th iteration. The last two terms represent the time mapping for the last iteration. The second term, $(s - 1) * E$ represents the time taken for all the paths in a single iteration except the last path. The third term, e gives the time mapping for the substeps of the last path. The total time required for the computation, T_{ex} is given by the maximum value of Π .

$$T_{ex} = \max_{i,s,e} \Pi$$

Substituting the maximum limits of each parameter in equation 5, we get

$$T_{ex} = (N - 1) * S * E + (S - 1) * E + E$$

The total execution time is thus given by

$$T_{ex} = N * S * E \quad (6)$$

The above equation represents the total time required to execute N iterations of S statements with E substeps.

II.5.2 Multiprocessor no Pipeline Method (MNP)

This method uses several processors to perform the given loop computation. Defining total number of processors to be K, ($1 \leq p \leq K$) two cases can be considered. If the number of statements is less than or equal to number of processors ($S \leq K$), each statement is assigned to a single processor. This is called a no wrap-around case. Several statements can be assigned to each processor if the number of statements is greater than the number of processors, ($S > K$). This is called as a wrap-around case. This technique might be called the statement(s) per processor method. Multiple Instruction-Multiple Datastream (MIMD) architectures use this kind of technique [5], where each instruction stream as well as data stream is processed by a single processing element. Each loop statement corresponds to an instruction stream. Considering no wrap-around of statements we can represent the processor mapping P as

$$P(i, s, e) = s$$

The time mapping function, Π , is given by

$$\Pi(i, s, e) = (i - 1) * E + e \quad (7)$$

The equation representing function Π consists of two terms, the first term represents the time required to execute a single iteration and the second term represents the time mapping for the substeps of the last iteration. The total execution time T_{ex} , is given in terms of maximum value of Π .

$$T_{ex} = \max_{i,s,e} \Pi$$

Replacing the maximum limits of the parameters in equation 7, we get

$$T_{ex} = N * E \quad (8)$$

The above equation represents total execution time for the loop code having S statements, each of length E over N iterations for the no wrap-around case.

Using the same approach for the wrap-around of paths ($S > K$), the total execution time T_{ex} , for the MNP method is given by

$$T_{ex} = N * E \left\lceil \frac{S}{K} \right\rceil \quad (9)$$

The above equation represents the time required to execute S instructions, each of length E over N iterations for both the wrap-around and no wrap-around cases.

II.5.3 Multiprocessor Individual Pipeline Method (MIP)

This method is an example of fine grain parallelism with each node (step in the statement) assigned to a single processor. For this case each statement uses its own pipeline. This method might be called the step-per-processor method and is used in systolic systems[5]. A systolic system consists of a set of interconnected cells, each capable of performing some simple operation. Information in a systolic system flows between cells in a pipelined fashion. The processor mapping function, P , for this method is given by

$$P(i, s, e) = (s - 1) * E + e$$

The time mapping function, Π , for this method is given by

$$\Pi(i, s, e) = E - 1 + i \quad (10)$$

The total execution time T_{ex} is given by the maximum value of Π .

$$T_{ex} = \max_{i,s,e} \Pi$$

Substituting the maximum value of each parameter in equation 10 gives

$$T_{ex} = E - 1 + N \quad (11)$$

The above equation represents the total time required to execute the given loop code over N iterations using the MIP method.

The above mapping methods will be compared with our new method. The uniprocessor method is the most general method and does not provide any advantage in terms of parallelism. It is independent of the grain size of nodes. The MNP method considers the partitioning at the statement level. The MIP method considers the computation at a fine grain size.

In practice there are many other mapping methods such as iteration per processor, iterations per processor, etc. These methods map an iteration or iterations to each processors. These methods do not consider any intraloop parallelism in the mapping process. Therefore these methods offer coarse grain parallelism for the given computation. Our research focuses on intraloop parallelism, which considers the parallelism at a finer grain level. Other fine grain methods such as a separate processor for each node for each individual iteration can be considered. However these methods require a large number of processors. Therefore these methods are too expensive to be practical.

II.6 MOTIVATION FOR OUR RESEARCH

In Chapter III, we propose a loop transformation method which maps the given computation space to a limited number of processors in a pipelined fashion. The motivation for this idea has arisen from the following facts.

The first important point is the level of parallelism considered in the mapping process. Partitioning at a fine grain level offers more parallelism in the mapping process. Earlier researchers have tried partitioning on the iteration level. The iterations are partitioned considering the dependences between them and these partitions are assigned to the available processors. No intraloop parallelism was involved in the mapping process. In other words, the level of parallelism was not significantly high in the given mapping pro-

cess.

This parallelism can be greatly increased by considering the internal partitioning of an iteration. Partitioning can be performed on the fine grain level such as a single step in the statement. These fine grain partitions can be mapped to the given processor space. We have tried to explore the intraloop parallelism and considered fine grain partitioning for our proposed transformation method.

Second, the use of pipelining improves the system performance by overlapping the execution of several different instructions. When there is no interaction among the instructions, several instructions can be in different stages of execution simultaneously, making the execution process faster. Thus a high degree of overlapped parallelism is possible by pipelining a small number of processors.

The third important point is that the number of processors required by a mapping must be considered. The statement per processor method (MNP) assigns a loop statement to an individual processor. Thus the processor space in this case depends on the number of statements in the loop code. This method calls for more processors as the number of loop statements increases. The step-per-processor method (MIP) maps an individual sub-step in the loop statement to a processor. The processor space in this case is a function of number of statements as well as the length of each individual statement. Thus for these two methods, the number of processors may rapidly become impractical.

A method that can achieve good performance with few processors is highly desired. The uniprocessor scheme is not a solution to this problem as it does not offer any parallelism. Therefore a mapping method which can provide a compromise solution is needed. The multiplexed pipeline method (MUP) meets this requirements. It is described in detail in the next Chapter.

CHAPTER III

PERFORMANCE EVALUATION FOR MULTIPLEXED PIPELINE METHOD

III.1 MULTIPLEXED PIPELINE MAPPING

Recall that the computation space shown in Figure 5 consists of two main components. These components are loop control statement and processing statements. In what follows, these components are referred to as paths. Each of these path is made up of substeps with a sequential precedence relation between substeps. The multiplexed pipeline transformation maps the loop to processors arranged in a pipeline.

PS4			$S_{1,4}$	$S_{2,4}$	$S_{3,4}$	-----		$S_{s,4}$
PS3		$S_{1,3}$	$S_{2,3}$	$S_{3,3}$	-----		$S_{s,3}$	
PS2	$S_{1,2}$	$S_{2,2}$	$S_{3,2}$	-----		$S_{s,2}$		
PS1	$S_{1,1}$	$S_{2,1}$	$S_{3,1}$	-----	$S_{s,1}$			

Figure 6. Multiplexed Pipeline Mapping.

Figure 6 shows a Gantt chart illustrating the execution of a loop mapped via multiplexed pipelining. $S_{i,j}$ denotes substep j of path i . PS_j is the j th pipeline stage or processor. As evident from the figure, each stage of the pipeline contains a substep from each path. The mapping can be informally described as follows.

Substep j of each path is assigned to processor PS_j . If there are more substeps per path than there are processors, the assignment wraps around to the first processor. In other words, for K processors, substep $K+1$ is assigned to processor 1 and so on. At each processor, the substeps are executed in same order that obeys the dependence constraints of the program.

As seen from Figure 6, there exist various ways in which the loop paths can be ordered for execution. This can be explained with respect to the mapping functions Π and P . Since the path lengths are assumed equal, for different ordering of loop paths, the processor mapping function P is fixed. However for different ordering of loop paths, the time mapping function Π may vary. Therefore it is a necessary task to evaluate the effect of the ordering of loop paths on Π and on execution time. The analysis below fixes P and considers the effect of these ordering on the time mapping function Π and tries to find an ordering which will result in minimum execution time.

III.2 EXPLORING THE SOLUTION SPACE

For this analysis, the solution space must first be described. Each ordering of loop paths is a point in this solution space. For the loop computation space having S paths, there are S substeps at the first processor. If we assume the same ordering is used at all processors, the total number of orderings becomes $S!$. As S grows, the solution space quickly becomes enormous. To simplify analysis, we group the loop components into two classes. These classes can be arranged in various permutations to cover the total solution space. These classes are:

1. LC : This class contains the loop control statement.
2. TH : This class contains all the threads (processing statements).

The loop control statement plays an important role in execution. We consider the permutations of threads relative to the placement of the loop control statement. Based on

the placement of the loop control statement with respect to the threads, solution subspaces are defined. The notation ORD below represents all possible orderings of threads within that solution subspace. The subspaces are:

1. Loop control statement at the beginning followed by all arrangements of threads and denoted LC ORD(TH). (LCF subspace)
2. Loop control statement at the end preceded by all arrangements of threads and denoted ORD(TH) LC. (LCE subspace)
3. Loop control statement in between threads. (LCB subspace) This subspace contains all the permutations from the total solution space excluding the above two subspaces. This subspace can be denoted by $LCB = \{ O \mid O \text{ is the path ordering with loop control statement in between threads } \}$

We next generate a model which will predict the execution time of the given loop. Recall that the following variables, first defined in section II.4, describe the computation space for a loop:

1. N : Total number of iterations.
2. E : Total number of substeps in a path.
3. K : Total number of processors used for computation.
4. S : Total number of paths in the loop(including the loop control statement).

The general function for the execution time of the loop can be represented as

$$T_{ex} = F(ord(S_{1,1}, S_{2,1}, S_{3,1}, \dots, S_{n,1}), N, E, S, K)$$

where ord represents the ordering of the substeps of each path at the first processor.

To simplify analysis, we proceed as follows. We start with a simple first case where S, K and E are assumed to be equal. For this case, we develop a general model and then illustrate its performance for each of the three solution subspaces. The simple case

gives an intuitive feeling for the analysis performed while developing the execution models. This will help in understanding the model for the more complicated situation when all parameters are independent.

PS4				L	S1	S2	S3	L	S1	S2	S3					
PS3			L	S1	S2	S3	L	S1	S2	S3						
PS2		L	S1	S2	S3	L	S1	S2	S3							
PS1	L	S1	S2	S3	L	S1	S2	S3								

7(a) Gantt Chart for $K=S=E$, LCF

PS4				S1	L	S2	S3		S1	L	S2	S3				
PS3			S1	L	S2	S3		S1	L	S2	S3					
PS2		S1	L	S2	S3		S1	L	S2	S3						
PS1	S1	L	S2	S3		S1	L	S2	S3							

7(b) Gantt Chart for $K=S=E$, LCB

PS4				S1	S2	L	S3			S1	S2	L	S3			
PS3			S1	S2	L	S3			S1	S2	L	S3				
PS2		S1	S2	L	S3			S1	S2	L	S3					
PS1	S1	S2	L	S3			S1	S2	L	S3						

7(c) Gantt Chart for $K=S=E$, LCB

PS4				S1	S2	S3	L				S1	S2	S3	L		
PS3			S1	S2	S3	L				S1	S2	S3	L			
PS2		S1	S2	S3	L				S1	S2	S3	L				
PS1	S1	S2	S3	L				S1	S2	S3	L					

7(d) Gantt Chart for $K=S=E$, LCEFigure 7. First case $K = S = E$.

III.2.1 First Case : $K = S = E$

Figure 7 shows Gantt charts for a loop with the number of paths, the number of substeps per path and the number of processing elements equal. L represents the execution of the loop control statement in the pipeline stages. S1, S2, S3 etc., represent the execution of processing statements in the pipeline stages. PS1, PS2, PS3 etc., represent the pipeline stages or processors. There are four Gantt charts, each showing a different position of the loop control statement with respect to the rest of the threads. Entry nodes of each path are assigned to the first stage of the pipeline execution.

The processor mapping, P , in this case is given by

$$P(i, s, e) = e$$

Since the multiplexed pipeline method uses pipelining, there is an overlap between the iterations in the pipe. This overlap has to be considered in the time mapping function, Π . Therefore the equations representing the time mapping for this method are of the form:

$$\Pi(i, s, e, \Delta) = (i - 1) * [T_1] - (i - 1) * [T_2 - \Delta] + [T_3]$$

where

T_1 = Time required for a single iteration without considering the overlap;

T_2 = Overlap time between iterations; and

T_3 = Time mapping for the substeps of the last iteration.

In this case, the time mapping is

$$\Pi(i, s, e, \Delta) = (i - 1) * (E - 1 + S) - (i - 1) * (E - 1 - \Delta) + e - 1 + s \quad (1)$$

The term T_1 means that E cycles are required to complete the first path and $S - 1$ cycles are required to complete the remaining paths. T_2 represents the possible overlap between the iterations. The value of this overlap ranges between 0 and $(K - 1)$. The term

Δ refers to the pipeline delay between iterations created by the loop control statement and the dependence between itself and other processing statements. The third term representing T_3 gives the time mapping for each substep for the i th iteration. The total execution time T_{ex} is given by the maximum value of function Π .

$$T_{ex} = \max_{i,s,e,\Delta} \Pi$$

Substituting the maximum values for the parameters and rearranging gives:

$$T_{ex} = [(E - 1) + (S + \Delta) * N - \Delta] \quad (2)$$

The above equation shows that it takes $(E - 1)$ cycles to fill up the pipeline and each iteration is completed after $(S + \Delta)$ cycles. The above equation represents all positions of the loop control statement, i.e., all threads orderings. The range of Δ determines the solution subspace, i.e., LCF, LCE, LCB.

III.2.1.1 LCF Subspace. The Gantt chart for this case is shown in figure 7(a). The general model for execution time was

$$T_{ex} = [(E - 1) + (S + \Delta) * N - \Delta]$$

For the LCF subspace, Δ is zero. Thus the equation can be rewritten as

$$T_{ex} = [(E - 1) + S * N] \quad (3)$$

This is a normal pipeline equation which resembles equation 4 in Chapter II. Once the pipeline is full (after $E - 1$ cycles), then each iteration is completed after S cycles. To complete N iteration requires $S * N$ cycles. Thus, LCF is fully pipelined with no delay between the iterations.

To verify that this model is correct for all points in the LCF solution subspace, consider Figure 7(a). It can be seen that changing the relative positions of the threads within the TH group does not change the execution time since the threads are identical in

length.

III.2.1.2 LCE Subspace. The Gantt chart for this case is shown in Figure 7(d). The general model for execution time was

$$T_{ex} = [(E - 1) + (S + \Delta) * N - \Delta]$$

For this subspace, Δ is $(S - 1)$ and the execution time is

$$T_{ex} = [(E - 1) + (S + S - 1) * N - (S - 1)]$$

Since $E = S$, we have

$$T_{ex} = [(E - 1 + S) * N] \quad (4)$$

This shows that the pipeline is filled and drained for each iteration, i.e., there is no overlap between iterations. In other words it means that once the pipelined is filled after $E - 1$ cycles, it is drained for S cycles before it is filled again. Again, since the threads are identical, this model is valid for all orders of the TH group.

III.2.1.3 LCB Subspace. The Gantt chart for this case is shown Figure 7(b) and Figure 7(c). The general model for execution time was

$$T_{ex} = [(E - 1) + (S + \Delta) * N - \Delta] \quad (5)$$

The value of pipeline delay Δ in LCB subspace varies from 1 to $S - 2$, with larger Δ giving higher execution time. Equation 5 describes the cases with some pipeline delay and some overlap between iterations.

III.2.1.4 Summary for the First Case ($S = E = K$). For this simple case we can say that,

$$T_{ex} (\Delta = 0) \leq T_{ex} (0 < \Delta < S - 1) \leq T_{ex} (\Delta = S - 1)$$

or

$$T_{ex_{lcf}} \leq T_{ex_{lcb}} \leq T_{ex_{lce}}$$

Thus execution time increases with Δ , i.e., as the loop control statement is moved toward the end of the ordering.

III.2.2 General Case : All Parameters Varied

In this case, all the parameters are varied independently resulting in a general model for multiplexed pipelining. The processor mapping function, P , is given by

$$P(i, s, e) = (e - 1) \bmod K + 1$$

The time mapping function, Π , is given by

$$\Pi(i, s, e, \Delta) = (i - 1) * [T_1] - (i - 1) * [T_2 - \Delta] + [T_3]$$

where

T_1 = Time required for a single iteration without considering the overlap;

T_2 = Overlap time between iterations; and

T_3 = Time mapping for substeps of the last iteration.

Substituting for T_3 gives:

$$\Pi = (i - 1) * [T_1] - (i - 1) * [T_2 - \Delta] + [(e - 1) \bmod K + s + \left\lfloor \frac{e - 1}{K} \right\rfloor * \max(K, S)]$$

where

$$T_1 = (E + S - 1) \text{ for } S \leq K;$$

$$T_1 = [K - 1 + \left\lfloor \frac{E * S}{K} \right\rfloor] \text{ for } S > K \text{ and } E/K \text{ is an integer;}$$

$$T_1 = \left[\left\lfloor \frac{E}{K} \right\rfloor * S + E \bmod K - 1 \right] \text{ for } S > K \text{ and } E/K \text{ is not an integer;}$$

$$T_2 = [S - 1] \text{ for } S \leq (E - 1) \bmod K; \text{ and}$$

$$T_2 = [(E - 1) \bmod K] \text{ for } S > (E - 1) \bmod K.$$

There are three distinct equations for the term T_1 . In the first case, the number of

paths in the loop is less than or equal to number of processors. This T_1 resembles the basic pipeline equation where the first path takes E cycles to complete and the remaining paths are finished in $S - 1$ cycles.

In the second case, the number of paths is greater than the number of processors and E/K is an integer. The first term of T_1 , $K - 1$, is the number of cycles required to fill the pipeline and the second term, is the number of cycles required to finish the paths, after the pipeline is full.

In the third case, the number of paths is again greater than the number of processors and E/K is non-integer. The equation representing the third subcase is shown above. This case is similar to the previous case, except that a path may finish without passing through all the stages of the pipeline.

Execution time (T_{ex}) for the loop is equal to the maximum value of Π and is given by

$$T_{ex} = \max_{i,s,e,\Delta} \Pi = N * [T_1] - (N - 1) * [T_2 - \Delta] \quad (6)$$

T_1 represents the time for a single iteration without any overlap and T_2 stands for the maximum overlap between the iterations. These terms are shown below for different conditions.

$$T_1 = (E + S - 1) \text{ for } S \leq K;$$

$$T_1 = [K - 1 + [\frac{E * S}{K}]] \text{ for } S > K \text{ and } E/K \text{ is an integer;}$$

$$T_1 = [\left[\frac{E}{K} \right] * S + E \bmod K - 1] \text{ for } S > K \text{ and } E/K \text{ is not an integer;}$$

$$T_2 = [S - 1] \text{ for } S \leq (E - 1) \bmod K; \text{ and}$$

$$T_2 = [(E - 1) \bmod K] \text{ for } S > (E - 1) \bmod K.$$

Again we illustrate the general equation for each of the three solution subspaces.

III.2.2.1 LCF Subspace. Figure 8 shows the Gantt chart for the LCF subspace.

PS4				L	S1	S2	S3	S4							
PS3			L	S1	S2	S3	S4	L	S1	S2	S3	S4			
PS2		L	S1	S2	S3	S4	L	S1	S2	S3	S4				
PS1	L	S1	S2	S3	S4	L	S1	S2	S3	S4					

Figure 8.General case LCF subspace.

The general model for execution time is

$$T_{ex} = N * [T_1] - (N - 1) * [T_2 - \Delta]$$

The pipeline delay Δ is zero for this subspace. Therefore the execution time is:

$$T_{ex} = N * [T_1] - (N - 1) * [T_2] \quad (7)$$

The LCF subspace has the minimum execution time as compared to other two solution subspaces.

III.2.2.2 LCE Subspace. Figure 9 shows a sample Gantt chart for LCE subspace.

PS4				S1	S2	S3	S4	L							
PS3			S1	S2	S3	S4	L	S1	S2	S3	S4	L			
PS2		S1	S2	S3	S4	L	S1	S2	S3	S4	L				
PS1	S1	S2	S3	S4	L	S1	S2	S3	S4	L					

Figure 9.General case LCE subspace.

The general execution time model is

$$T_{ex} = N * [T_1] - (N - 1) * [T_2 - \Delta]$$

For this subspace the loop control statement executes after the other processing statements. Thus the pipeline delay, Δ achieves the value of T_2 . Thus the execution time is

$$T_{ex} = N * [T_1] \quad (8)$$

The LCE subspace has the worst execution time of all the three solution subspaces due to no overlap between iterations.

III.2.2.3 LCB Subspace. The general execution model is again

$$T_{ex} = N * [T_1] - (N - 1) * [T_2 - \Delta] \quad (9)$$

The value of pipeline delay Δ in the LCB subspace increases from 1 to $T_2 - 1$. Thus the execution time in the LCB subspace lies between the LCF and LCE subspaces.

III.2.2.4 Summary for the General Case. Considering the models for each of the solution subspaces, for a general loop with S paths, each path having E substeps, with iteration count N and executed on K processors we can say that,

$$T_{ex_{lcf}} \leq T_{ex_{lcb}} \leq T_{ex_{lce}}$$

The LCF category begins with the execution of the loop control statement at the leftmost position in the Gantt chart. The loop control statement is responsible for the generation of future iterations. The rightward motion of the loop control statement introduces delays for future iterations. The maximum delay occurs when the loop control statement is at the end, i.e., in the LCE subspace.

The execution models developed for the multiplexed pipeline method reflects this behavior. In the LCF subspace, the value of pipeline delay Δ is always zero, which corresponds to the minimum execution time obtained for the multiplexed method. This delay increases for LCB and LCE subspaces, increasing the execution times for those

subspaces.

III.3 COMPARISON OF MAPPING METHODS

In this section we compare the mapping methods developed in Chapter II and III. The mapping methods were compared with respect to the following performance measures:

1. Cost: This is the number of processors used.
2. Execution Time.
3. Flexibility : This refers to the usability of the mapping method for a wide range of loops.

III.3.1 Uniprocessor and Multiprocessor Methods

The uniprocessor mapping method maps loop code of any size to a single processor. This method wins in terms of cost. However it loses with respect to the execution time except when

$$S = 1 \text{ and } E = 1.$$

In this case no parallelism is possible. Loops of much bigger sizes will be encountered in the practical case. Thus for the loop sizes where $S > 1$ and $E > 1$, the multiprocessor mapping schemes can be considered for better execution time.

III.3.2 Between MNP and MIP Method

MIP is a fine grain method in terms of parallelism and yields the best execution time. Comparing with MNP we get:

$$\begin{aligned} T_{mnp} &> T_{mip} \\ N E &> E - 1 + N \\ (N-1) E &> (N-1) \\ E &> 1 \end{aligned}$$

Therefore for $E > 1$

$$T_{mip} < T_{mnp}$$

In other words, MIP is faster whenever paths have more than one substep. For a loop with path lengths greater than 1, MIP uses more processors than MNP. Therefore, when $E > 1$, MIP is faster but costlier. For the case where $E = 1$, the two methods are identical.

III.3.3 Between MUP and MNP Method

The multiplexed pipeline method uses a fixed number of processors, K independent of the loop size parameters. The MNP method with no wrap-around uses S processors. Under these conditions:

1. If $S > K$, the MNP is faster by a factor of S/K . For this case, MUP is cheaper than MNP by a difference of $(S - K)$ processors.
2. If $S \leq K$, the MNP is better than MUP in terms of both execution time and cost.

Given a fixed number of processors K , for both methods, several cases must be considered.

III.3.3.1 $E > K$, $S > K$, S/K not an integer, E/K an integer. Under these conditions, when

$$N E > \frac{K(K-1)}{K - (S \bmod K)}$$

MUP has lower execution time than the MNP method.

III.3.3.2 $E > K$, $S > K$, S/K not an integer, E/K not an integer. Under these conditions, when

$$N E > \frac{(K - (E \bmod K)) N S + ((E \bmod K) - 1) K}{K - (S \bmod K)}$$

MUP has lower execution time than the MNP method.

III.3.3.3 $E > K, S > K, S/K$ an integer. Under these conditions, MNP always has lower execution time than the MUP method.

III.3.3.4 $E > K, S \leq K$. Under these conditions, MNP always has lower execution time than the MUP method.

III.3.3.5 $E = K, S > K, S/K$ not an integer. Under these conditions, when

$$N > \frac{E - 1}{K - (S \bmod K)}$$

MUP has lower execution time than the MNP method.

III.3.3.6 $E = K, S > K, S/K$ an integer. Under these conditions, MNP always has lower execution time than the MUP method.

III.3.3.7 $E = K, S \leq K$. Under these conditions, when,

$$N E > K - 1 + N S$$

MUP has lower execution time than the MNP method.

III.3.3.8 $E < K, S > K, S/K$ not an integer. Under these conditions, when,

$$N E > \frac{(E - 1) K}{K - (S \bmod K)}$$

MUP has lower execution time than the MNP method.

III.3.3.9 $E < K, S > K, S/K$ an integer. Under these conditions, MNP always has lower execution time than the MUP method.

III.3.3.10 $E < K, S \leq K$. Under these conditions, when

$$N E > E - 1 + N S$$

MUP has lower execution time than the MNP method.

Therefore, for fixed cost, MUP shows better performance in terms of execution time as compared to MNP method and remains a suitable choice under those loop size conditions.

TABLE I
LOWER EXECUTION TIME BETWEEN MNP AND MUP FOR $E > K$

$E > K$		
$S > K$		$S \leq K$
$S/K = \text{integer}$	$S/K = \text{non-integer}$	
MNP	$N E > \frac{K(K-1)}{K - (S \bmod K)}$ $N E > \frac{(K - (E \bmod K))N S + ((E \bmod K) - 1)K}{K - (S \bmod K)}$	MNP

TABLE II
LOWER EXECUTION TIME BETWEEN MNP AND MUP FOR $E = K$

$E = K$		
$S > K$		$S \leq K$
$S/K = \text{integer}$	$S/K = \text{non-integer}$	
MNP	MUP if $N > \frac{E-1}{K - (S \bmod K)}$	MUP if $N E > K - 1 + N S$

TABLE III
LOWER EXECUTION TIME BETWEEN MNP AND MUP FOR $E < K$

$E < K$		
$S > K$		$S \leq K$
$S/K = \text{integer}$	$S/K = \text{non-integer}$	
MNP	MUP if $N E > \frac{(E-1)K}{K - (S \bmod K)}$	MUP if $N E > E - 1 + N S$

Table I, II and III summarizes the comparison between MNP and MUP method for the conditions $E > K$, $E = K$ and $E < K$ respectively. From the comparison of the

MUP and MNP method, it is seen that MUP method has lower execution time than the MNP method for various loop size conditions shown above. Under these conditions, MUP method is a suitable choice.

III.3.4 Between MUP and MIP Method

The MIP method uses more processors compared to MNP and MUP methods. The number of processors required for MIP is the product of number of paths and length of each path ($S * E$). For MIP, wrap-around of a loop path can take place lengthwise if number of processors are less than ($S * E$). The MIP method can be defined as step(s) per processor method with individual pipeline for each path.

Given the loop, the choice between MUP and MIP method can be made as shown below. Two types of comparisons are shown. The first type assumes unlimited processors while the second type assumes fixed number of processors. Considering unlimited processors, the comparison between MUP and MIP can be made as follows. MUP uses K processors independent of loop size.

1. If $S * E > K$, the MIP is faster by a factor of $\frac{E * S}{K}$ than MUP method. In this case MUP is cheaper than MIP method by a difference of $(E * S - K)$ processors.
2. If $S * E \leq K$, MIP is better than MUP in terms of both execution time and cost.

Given fixed number of processors, comparison between MUP and MIP is dependent upon size of the loop. In case of a wrap-around, there is a bound on the number of processors used for MIP method. The minimum number of processors MIP method requires in case of a wrap-around is $S * n$, where $1 < n < E$. The fixed processors M in this case is

$$M = S * n$$

Under this condition, two cases can be considered.

- A. In case, when S is less than M and the value of variable n is between 1 and E , MIP has lower execution time than MUP method.
- B. In case, when $n = 1$ or $n < 1$, which means that $S = M$ or $S > M$, then MIP method is no longer valid for operation, as the step(s) per processor allocation is no longer true. In this case, MUP turns out to be more flexible as compared to MIP. Therefore for this case, MUP method is the valid choice.

In order to summarize the above comparison, definitions of all methods can be referenced.

1. Uniprocessor method uses a single processor.
2. MNP method uses S processors. In case of fewer processors, paths are wrapped around the processors.
3. MIP method uses $S * E$ processors and avails the facility of individual pipeline for each loop path. MIP considers the wrap-around of paths lengthwise if number of processors is less than $S * E$. The minimum number of processors required by MIP is $S * n$, where $1 < n < E$.
4. MUP method uses K processors independent of the loop size.

III.3.5 Summary of Comparison

MUP is compared with UM, MNP and MIP methods. MUP loses with respect to cost, but wins in terms of execution time when compared to UM. While comparing MUP with MNP and MIP, two types of comparisons were taken into consideration. The first type assumed unlimited processors, while the second type assumed fixed processors. In case when number of processors is unlimited, MUP justifies its value in terms of cost as compared to MNP method. When number of processors is fixed, MUP justifies its value in terms of execution time compared to MNP method for various loop size conditions. Thus for the fixed cost, MUP method proves itself to be better in terms of execution time

than MNP method.

In case when number of processors are unlimited, MUP method justifies its value in terms of cost as compared to MIP method. When number of processors are fixed, MIP has a lower execution time than MUP for certain loop size conditions. However, for other conditions MUP proves itself to be flexible in terms of usability for a wide range of loop sizes.

The above comparison considered the size of computation. For different computation sizes, other methods fail in terms of any of the three performance measures mentioned above. MUP tries to fill the slot of this failure with respect to these performance measures. Finally summarizing the comparison we conclude that MUP method justifies its value in terms of cost of operation , execution time and flexibility compared to rest of the mapping methods and remains a suitable choice.

CHAPTER IV

EXPERIMENTAL RESULTS AND ANALYSIS

IV.1 TYPES OF EXPERIMENTS

To verify the execution models for the multiplexed pipeline method, experiments were designed and performed. The execution models give the execution time in terms of number of time units. The individual time unit is the time to complete a pipeline stage. Thus to predict execution time, the measurement of this stage time was necessary.

Three kinds of experiments were performed. The first set measured pipeline stage time. The second set measured the total multiprocessor execution time versus the pipeline delay Δ . Loops with iteration counts of 400, 800, 1200 and 1600 were considered. The third set of experiments measured uniprocessor execution time, which was used to calculate speedup.

IV.2 GRAPH MODEL FOR EXPERIMENTS

A suitable dataflow cyclic graph matching our loop model was chosen and used in the experiments. The general structure of graph is shown in Figure 10. The threads inside the loop control statement represent the processing statements. The circles represent the substep in the loop paths. The Figure shows the general program graph with parameter variation represented by arrows. To find pipeline stage time, graph with parameters, $E = 9$, $S = 3$ and $N = 50, 100, 200, 300$ and 500 was considered. To validate the model, graph with parameters, $E = 12$, $S = 7$ and $N = 400, 800, 1200$ and 1600 was considered. 6 processors were used for the experiment purpose, i.e., $K = 6$. To calculate speedup, the

same graphs were used with 1 processor.

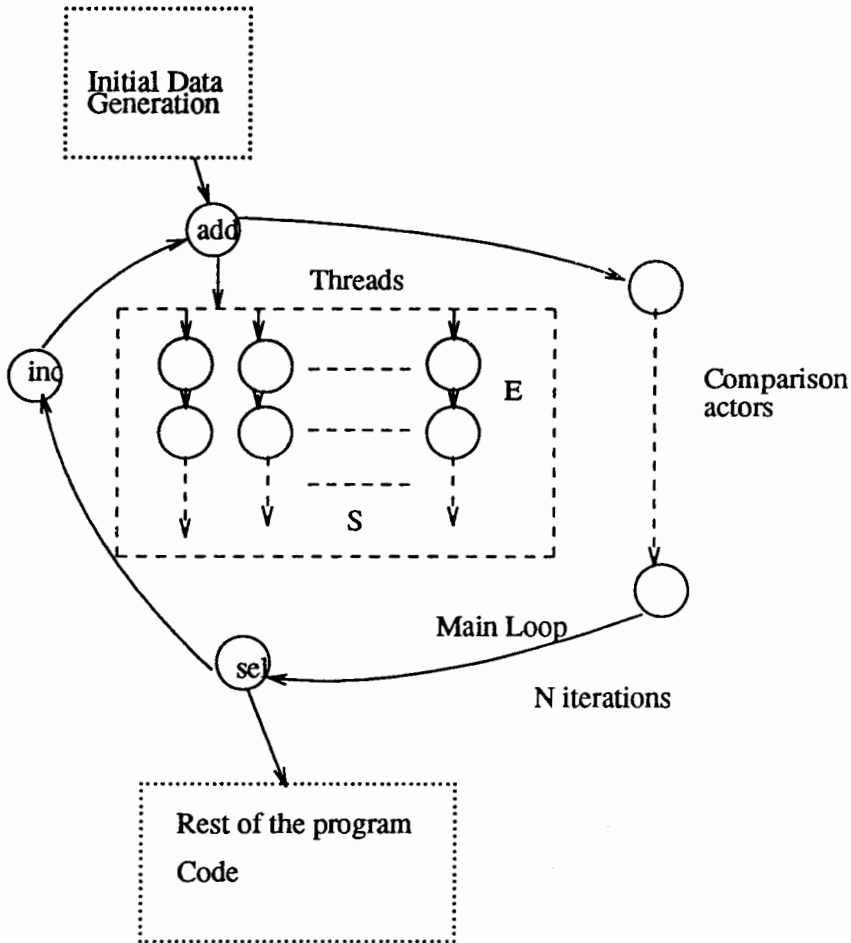


Figure 10. Graph Model for Experiments.

IV.3 THE EXPERIMENTAL SET UP

Experiments were run on Sun 3/50 workstations. The experiments were run at the midnight, when the system load is expected to be low. The wall clock time as opposed to CPU time was measured.

The ParPlum system, Jingsong[21] was used to conduct the experiments. The ParPlum mapping system consists of several stages. Its operation begins with reading the input graph and the architectural information. Using this information, the partitioning

algorithm divides the program graph into some number of modules. The partitioning information is passed to allocation algorithms which assign the existing program modules to specific processors. Scheduling decides the ordering of the tasks within these partitions and does the job of assigning priorities. Finally, the mapped program is executed under the control of a parallel interpreter for dataflow graphs. The partitioning, allocating and scheduling steps implement the multiplexed pipeline method with variable Δ .

IV.4 MEASUREMENT OF THE STAGE TIME

The models developed in Chapter III give total execution time in terms of the number of time units required for execution of the loop. This time unit is a single stage time in the execution. The stage time is a combination of the execution time for a single step and the communication time between the steps. Measurement of the stage time was needed to evaluate the models for the specific experiments used for validation.

The total number of stages for a single iteration of the sample graph were calculated using the execution model. The number of stages thus obtained was multiplied by the iteration count to calculate the total stages required for executing the loop. Five different iteration counts: 50, 100, 200, 300 and 500 were considered.

Each graph was then executed using the ParPlum mapping system. The execution time for each case was measured. The stage time was calculated by dividing the measured execution times by the total number of stages calculated via execution models. The stage time used ($T_s = 0.063$ second) was taken as the average stage time for the five values of iteration count. Table IV shows the data for this analysis.

The process can be more precisely explained as follows. Let

T_u = Number of units required for N iterations. (This is given by the models)

T_m = Measured execution time via experiment.

T_s = Pipeline stage time.

The three variables are related by the equation

$$T_m = T_u * T_s$$

Therefore, the stage time is given by

$$T_s = \frac{T_m}{T_u} \quad (1)$$

TABLE IV
CALCULATION OF STAGE TIME

N	50	100	200	300	500
T_u	502	1002	2002	3002	5002
T_m (seconds)	37.32	56.11	120.14	165.23	350.52
T_s (seconds)	0.074	0.056	0.060	0.055	0.070

In section IV.5, predicted execution time is calculated from the model and the stage time, i.e., $T_{pred} = T_{ex} * T_s$.

IV.5 EXECUTION TIME EXPERIMENTS

These experiments were run to compare the model with actual execution times. We were particularly interested in the effects of pipeline delay, Δ . The experiments used $E = 12$, $S = 7$, $K = 6$ and $\Delta = 0, 1, 2, 3, 4$ and 5 , and four different iteration counts. Table V and Table VI summarizes the results for 400 and 800 iterations while Table VII and Table VIII shows results for 1200 and 1600 iterations. The term Absolute Difference represents the difference between average measured value and predicted value.

TABLE V
EXECUTION TIMES FOR 400 ITERATIONS

For 400 iterations					
Δ	Measured(seconds)			Predicted(seconds)	Absolute Difference(seconds)
	Ave	Max	Min		
0	370	379	360	353	17
1	403	410	390	378	25
2	421,†	435	410	403	18
3	449	460	437	428	21
4	471	483	460	453	18
5	498	510	481	478	20

TABLE VI
EXECUTION TIMES FOR 800 ITERATIONS

For 800 iterations					
Δ	Measured(seconds)			Predicted(seconds)	Absolute Difference(seconds)
	Ave	Max	Min		
0	753	765	748	705	48
1	783	790	778	756	27
2	839	847	832	806	33
3	897	910	890	856	41
4	942,†	951	937	907	35
5	990	1020	970	957	33

TABLE VII
EXECUTION TIMES FOR 1200 ITERATIONS

For 1200 iterations					
Δ	Measured(seconds)			Predicted(seconds)	Absolute Difference(seconds)
	Ave	Max	Min		
0	1116	1128	1108	1058	58
1	1184	1194	1178	1134	50
2	1247	1258	1241	1209	38
3	1337	1350	1331	1285	52
4	1384	1391	1378	1360	24
5	1476	1491	1465	1436	40

TABLE VIII
EXECUTION TIMES FOR 1600 ITERATIONS

For 1600 iterations					
Δ	Measured(seconds)			Predicted(seconds)	Absolute Difference(seconds)
	Ave	Max	Min		
0	1505	1518	1496	1411	94
1	1586	1602	1572	1512	74
2	1677	1682	1674	1612	65
3	1765	1771	1759	1713	52
4	1850,†	1862	1840	1814	36
5	2005	2028	1998	1915	90

The system load was an important consideration while running the experiments. The experiments were run at midnight when the system load is expected to be low. Five runs were taken for each set of parameter values. For a few runs, the accuracy of the measured value was doubtful as compared to the rest of the runs for that particular point. For these suspicious runs, the run was discarded and another run was made to obtain five values for each point. The points indicated by † in Table V, Table VI and Table VIII each had one run discarded.

This process can be illustrated via the following example. One suspicious run was encountered for the iteration count ($N = 400$) and pipeline delay ($\Delta = 2$) as shown in Table V. The initial execution times measured for this case were 410, 435, 430, 971, 412. In this case, the execution time corresponding to fourth run ($T_m = 971$) was out of range as compared to other points. The run was repeated and the measured execution time was $T_m = 418$. The new value was in line with other measured values for the execution time. Therefor the suspicious value for the execution time was replaced with the new measured value and the average was taken. In total three runs were discarded.

Figure 11 plots measured and predicted execution time versus Δ for 400 and 800 iterations, while Figure 12 plots the same for 1200 and 1600 iterations. The solid line represents the measured values for the total execution times while the dashed line represents the predicted values for execution times. The solution subspaces are shown for each curve. For all the four cases the execution time increases as the value of Δ is increased. In other words, the best performance occurs when the outer loop is executed first ($\Delta = 0$). This is exactly the results predicted by the model. The predicted values are within 10 percent of the actual values and the curves are of the same general form. LCF, LCB and LCE represents the solution subspaces and are shown for each curve. In both cases, the optimum execution time is achieved under LCF solution subspace and the worst case execution time is obtained under LCE subspace.

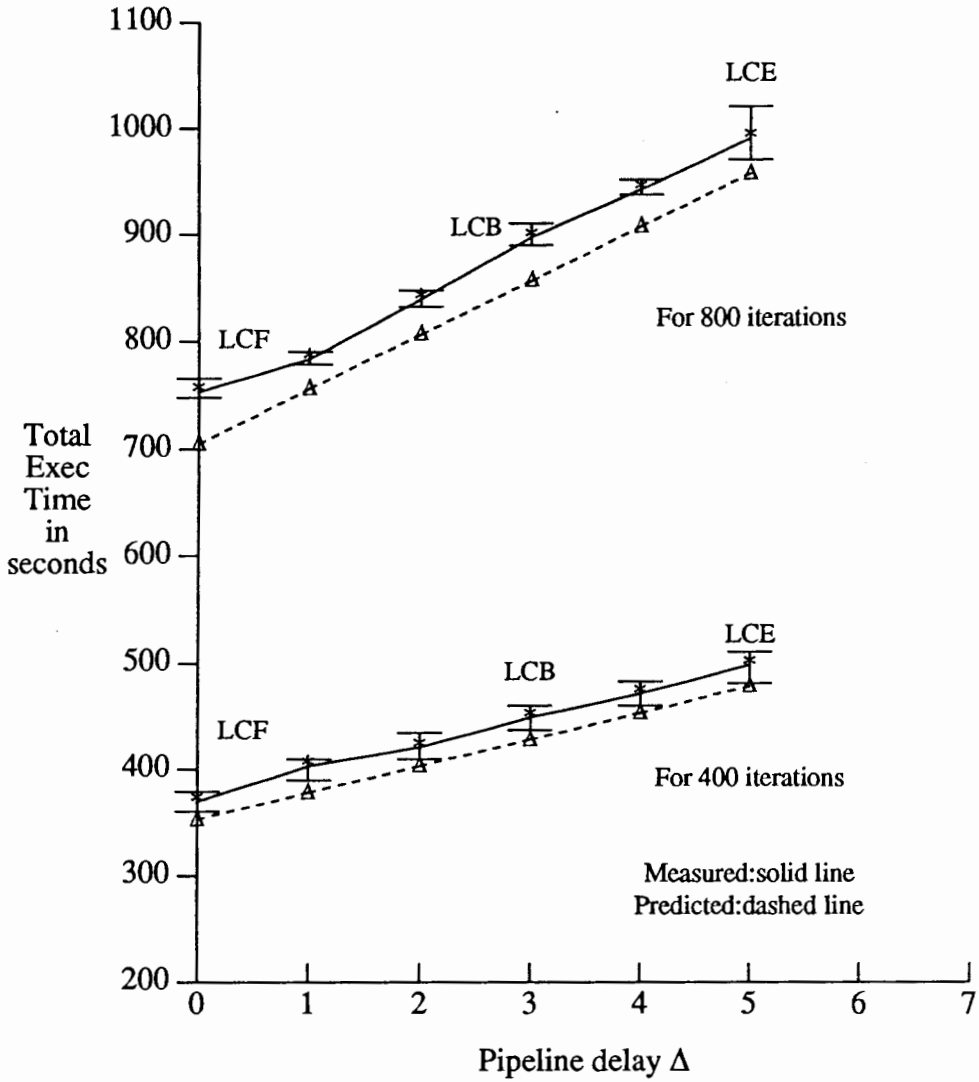


Figure 11. Execution Time for iterations 400 and 800.

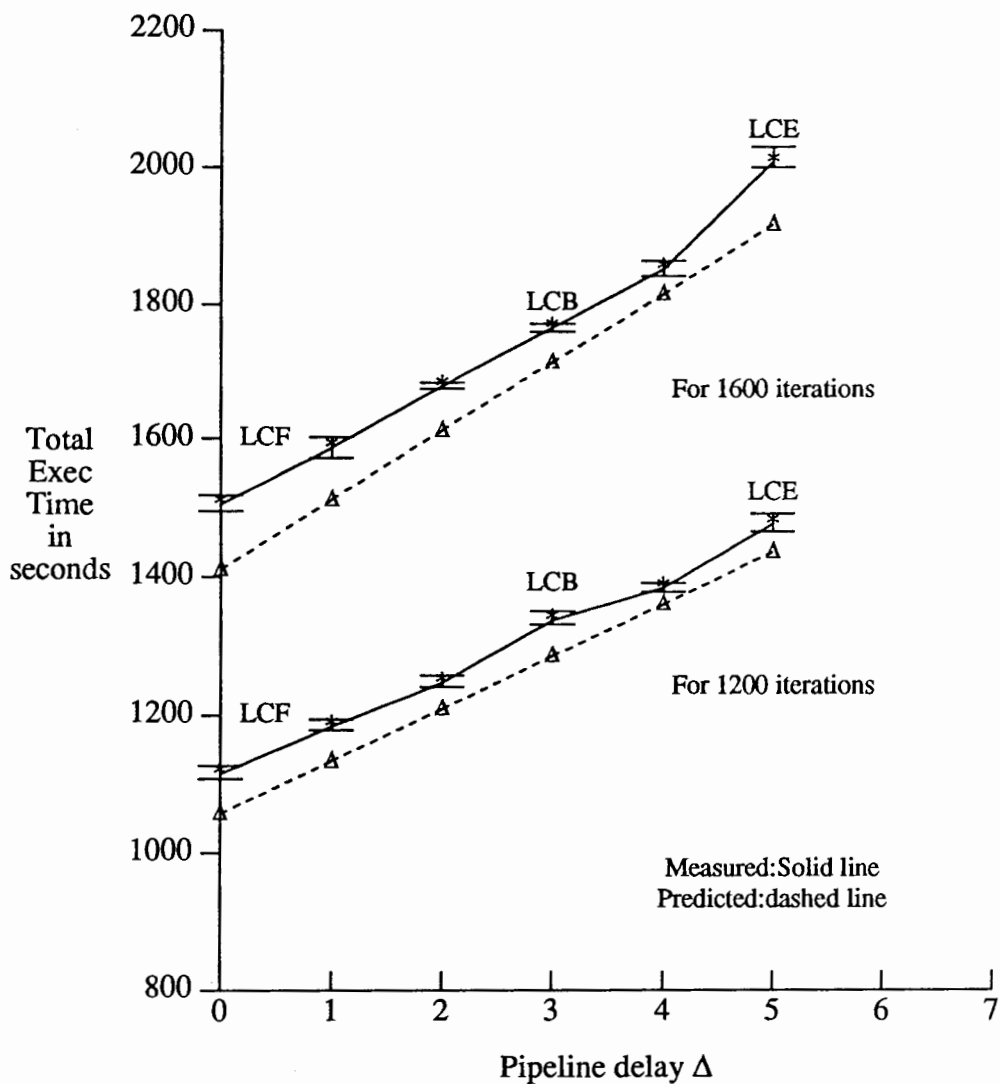


Figure 12. Execution Time for iterations 1200 and 1600.

Figure 13 plots the total execution time of the program versus the iteration count. Six curves are shown, one for each value of pipeline delay Δ . As expected, as the iteration count goes up, the total execution time goes up.

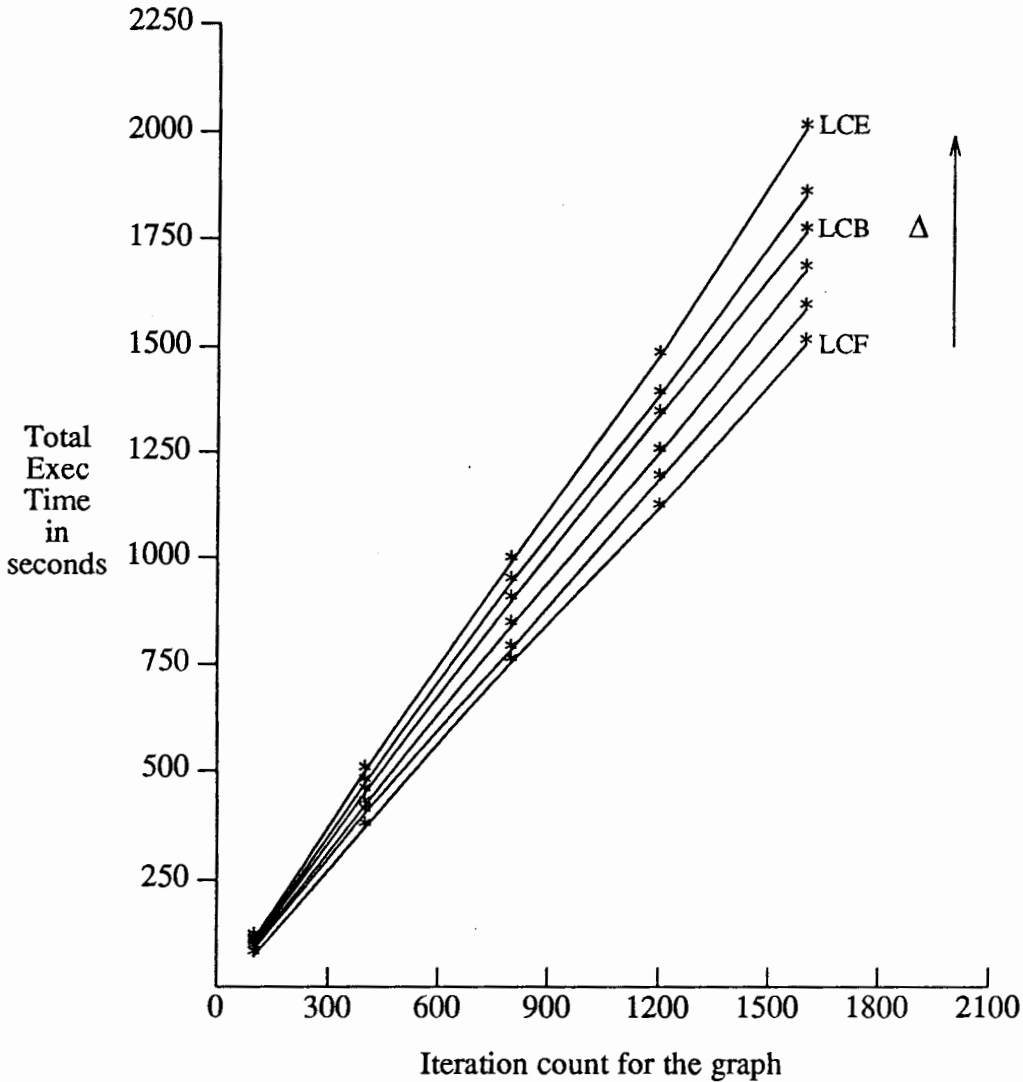


Figure 13. For $\Delta = 0, 1, 2, 3, 4$ and 5 .

IV.6 SPEEDUP MEASUREMENT

Speedup is defined as the ratio of the execution time of a given graph on a non-pipelined processor to the execution time on a pipelined processor with K stages. The

graphs were executed on a uniprocessor using the ParPlum mapping system, Jingsong[21] using a uniprocessor mapping and the corresponding execution times were measured. The graphs were identical with those used in section IV.5.(Note that Δ is not a parameter of the uniprocessor mapping.) Table IX shows the results.

TABLE IX
UNIPROCESSOR EXECUTION TIME(SECONDS)

	400 iterations	800 iterations	1200 iterations	1600 iterations
	1555	3216	4905	7365
	1560	3219	4911	7360
	1533	3224	4928	7391
	1513	3210	4920	7375
	1524	3231	4923	7379
Average	1537	3220	4917	7374

TABLE X
SPEEDUP RESULTS

Measured Speedup				
Δ	400 iterations	800 iterations	1200 iterations	1600 iterations
0	4.1	4.3	4.4	4.9
1	3.8	4.1	4.2	4.6
2	3.6	3.8	3.9	4.4
3	3.4	3.6	3.7	4.2
4	3.3	3.4	3.5	3.9
5	3.1	3.2	3.3	3.7

Table X shows the results for the measured speedups. Figure 14 shows the speedup curves for iteration counts of 400 and 800 while Figure 15 shows the speed up

curves for iteration counts of 1200 and 1600. Both figures show the pipelined speedup deteriorating as Δ is increased. Again the best speedup is achieved when the outer loop is executed first. Also note that the speedups range from 4.1 to 4.9 which is quite good compared with the theoretical maximum of 6.

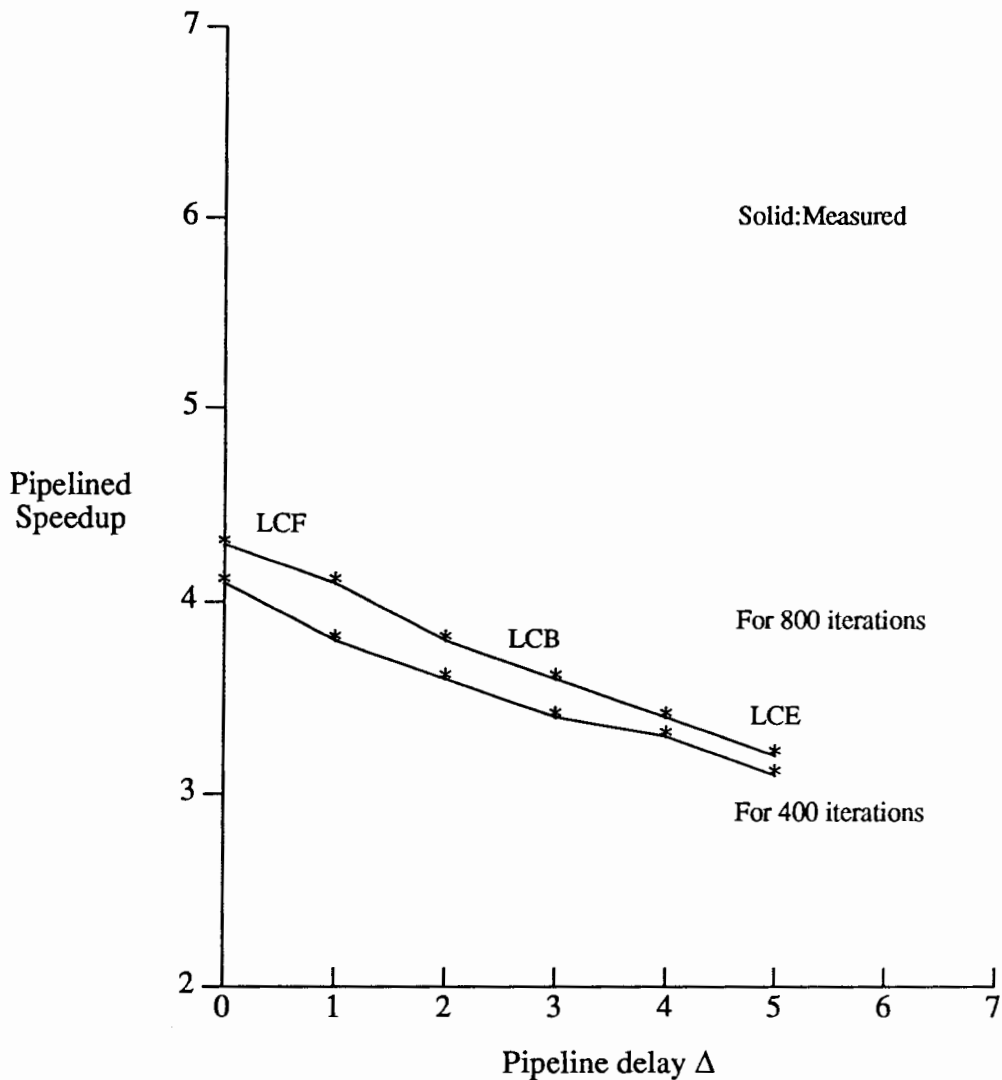


Figure 14. Pipelined Speedup for iterations 400 and 800.

Figure 15 shows the speedup curves for 1600 iterations. Again the maximum speedup is obtained under LCF subspace, where the pipeline delay Δ is zero.

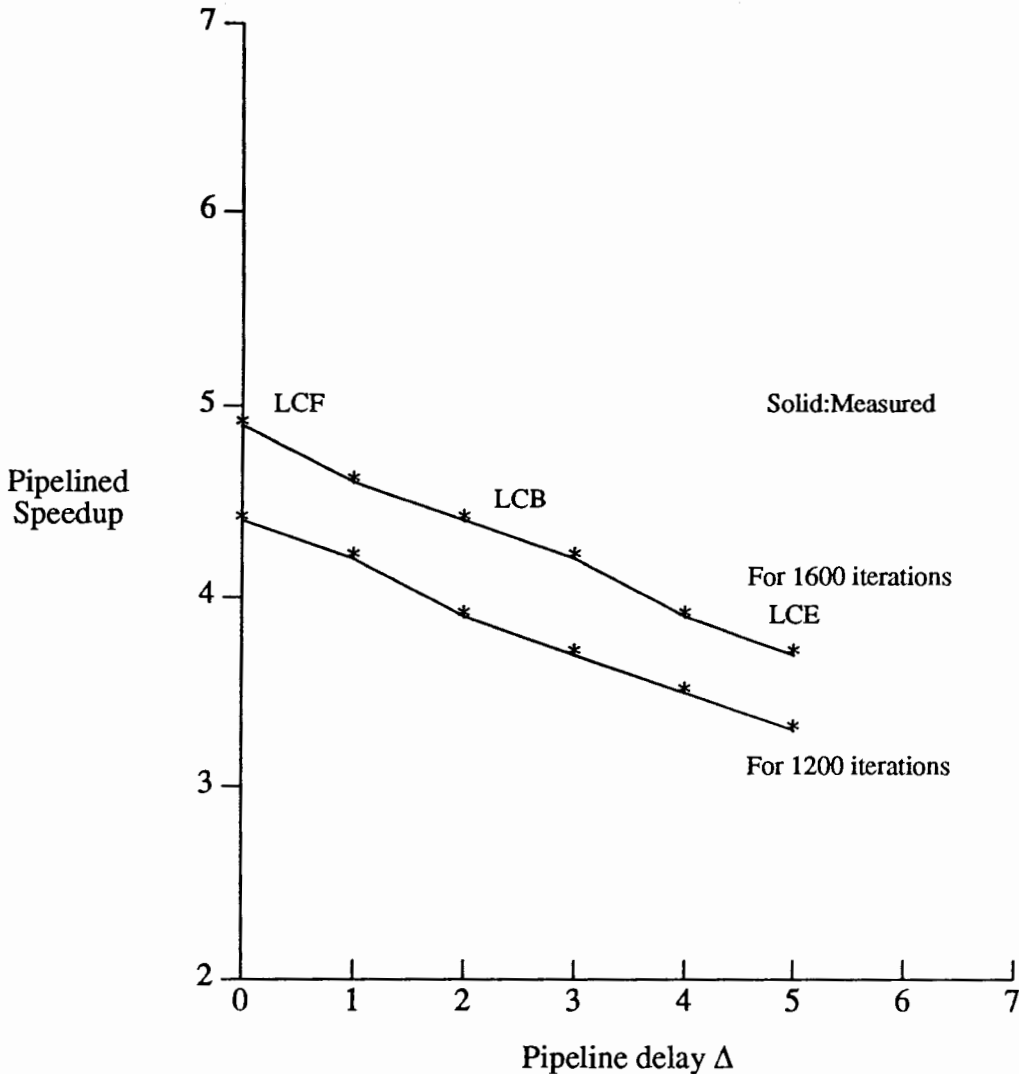


Figure 15. Pipelined Speedup for iterations 1200 and 1600.

IV.7 EXPERIMENTAL RESULT ANALYSIS

The results of experiments conducted to verify the execution models are described above. The models approximately predict execution time, matching inform, but consistently underpredicting. The predicted value consistently falls below the bounds of the experimental values which indicates a systematic error. To explain the error recall that the program graph shown in Figure 10 consists of three parts. The first part does the initial data generation for the loop. The second part contains the loop itself and the third

part consists of rest of the program. The execution time models only predict the execution time for the second part, but the measurements include all three parts of the program. In fact, the execution time for the first part of the graph should increase with iteration count. Table V, VI, VII and VIII show such an increase adding support to this explanation.

CHAPTER V

SUMMARY AND CONCLUSIONS

This thesis presented a loop transformation technique for scientific programs. Loops play an important role in compute-intensive programs. Loop transformation techniques can be viewed as a mapping process which maps the computation space to a processor space and time. A computation space consists of nodes corresponding to computation and arcs showing the flow of data or the dependence between nodes.

Two distinct functions define the space-time mapping for a given computation problem. Π defines the mapping from computation space to time space while P defines the mapping from computation space to processor space.

The size of a node defines the granularity of parallelism offered in the given mapping process. Earlier researchers have considered an iteration or a group of iterations as the grain size for their computation and thus ignored intraloop parallelism. The research presented in this thesis considers intraloop parallelism by looking at nodes at the fine grain level of substeps in the loop path.

Considering the computation space defined for our research, various mapping methods were considered for reference. Execution models were developed for these methods.

Next we defined the multiplexed pipeline method, its two mapping functions Π and P , and execution time models. The performance of the multiplexed pipeline method is affected by the ordering of the loop components. The models predicted that executing the loop control statement first would lead to the lowest execution time.

Experiments were conducted on a dataflow program graph containing loops to validate the developed models. The ParPlum mapping system was used for testing. The execution time experiments show that the optimal execution performance of the graph was obtained when the loop control statement was executed first, as predicted by the model.

The value of the multiplexed pipeline method can be explained in terms of three performance measures. These measures are cost of operation, execution time and flexibility. The MUP method loses in terms of cost of operation as compared to the uniprocessor (UM) method and wins in terms of execution time. Under various loop size conditions, the MUP method outperforms MNP and MIP with respect to cost, execution or both.

Also the execution models for the multiplexed pipeline method provide the advantage of mapping method selection at the compile time. The execution models were developed in terms of loop size parameters. Therefore, the user can decide on the MUP method before the execution of the program with the help of these parameters.

The research presented in this thesis considers the computation space of a loop which includes the loop control statement and processing statements. The considered computation space had several restrictions associated with its structure. These restrictions can be generalized in the future. The pipelined execution of our computation space is parallel to the concept of pipeline vector chaining. Pipeline chaining is a linking process that occurs when results obtained from one functional pipe are fed into another functional pipe. The computation space defined for this research can be considered for the vector processing applications in the future.

REFERENCES

- (1) Michael Wolfe, "The Tiny Loop Restructuring Tool," Proc. of the International Conference on Parallel Processing., vol. 2, Software, pp. 12-16, August 1991.
- (2) K.G. Kumar, D. Kulkarni, A Basu, "Generalized Unimodular Transformations for Distributed Memory Multiprocessors," Proc. of the International Conference on Parallel Processing., vol. 2, Software, pp. 146-149, August 1991.
- (3) P. tang, G. Michael, "Chain-Based Partitioning and Scheduling of Nested Loops for Multiprocessors," Proc. of the International Conference on Parallel Processing., vol. 2. Software, pp. 243-246, August 1991.
- (4) J. R. Allen, K. Kennedy, "Automatic Translation of Fortran Programs to vector form," ACM Transactions on Programming languages and Systems., vol. 9, pp. 491-542 October 1977.
- (5) K. Hwang, F. A. Briggs, "Computer Architecture and Parallel Processing," McGraw Hill, Inc 1984.
- (6) C.V.Ramamoorthy, "Pipeline Architecture," ACM Computing Surveys., vol. 9, Number 1, pp. 61-102, March 1977.
- (7) J. Robert Jump and S. Ahuja, "Effective Pipelining of Digital System," IEEE Transactions on Computers., vol. c-27, Number 9, pp. 855-865, September 1978.
- (8) Cooper R. G., "Distributed Pipeline," IEEE Transactions on Computers., vol. c-24, Number 7, pp. 1123-1132, November 1977.
- (9) Michael Wolfe, "Optimizing Supercompilers for Supercomputers," The MIT Press, Cambridge, Massachusetts 1989.
- (10) U. Banerjee, S. Chen, D. Kuck, R Towle, "Time and Parallel Processor Bounds for Fortran-Like Loops." IEEE Transactions on Computers., vol. c-28, Number 9, pp. 660-670, September 1977.
- (11) W. L. Miranker, A. Winkler, "Spacetime Representations of Computational structures," Computing 32, pp. 93-114, 1984.
- (12) D. I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," Proc. of the IEEE., vol. 71, Number 1, pp. 113-120, January 1983.

- (13) P. Lee, Z. Kedem, "Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays," *IEEE Transactions on Parallel and Distributed Systems.*, vol. 1, Number 4, pp. 64-76, January 1990.
- (14) C. King, W. chou, L. Ni, "Pipelined Data-parallel Algorithms:Part 1-Concepts and Modeling," *IEEE Transactions on Parallel and Distributed Systems.*, vol 1, Number 4, pp. 470-485, October 1990.
- (15) C. King, W. chou, L. Ni, "Pipelined Data-parallel Algorithms:Part 2-Design," *IEEE Transactions on Parallel and Distributed Systems.*, vol 1, Number 4, pp. 486-499, October 1990.
- (16) W. Shang, J.A.B. Fortes, "Independent Partitioning of Algorithms with Uniform dependencies," *IEEE Transactions on Parallel and Distributed Systems.*, vol. 41, Number 2, pp. 190-205, February 1992.
- (17) J.P.Sheu, C-Y. Chang, "Synthesizing Nested Loops Algorithms Using Nonlinear Transformation Method." vol. 2, Number 3, pp. 304-317, July 1991.
- (18) D. A. Patterson, J.H. Hennessy, "Computer Architecture-A Quantitative Approach," Morgan Kaufmann Publishers, Inc. 1990
- (19) J. Bruno, J. W. JonesIII, K. So, "Deterministic Scheduling with pipelined processors," *IEEE Transactions on Computers.*, vol. c-29, Number 4, pp. 308-316, April 1980.
- (20) J. P. Sheu and T. Tai, "Partitioning and Mapping Nested Loops on Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems.*, vol. 2, Number 4, pp. 430-439, October 1991.
- (21) F. Jingsong, "ParPlum: A System for Evaluating Parallel Program Optimization Methods," M.S Thesis, Department of Electrical Engineering, Portland State University, Portland, Oregon, August 1991.