View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCSE-2004-48

2004-08-04

Pipeline Task Scheduling with Appication to Network Processors

Seema Datar

Chip Multi-Processors (CMPs) are now available in a variety of systems and provide the opportunity for achieving high computational performance by exploiting application-level parallelism. In the communications environment, network processors (NPs), designed around CMP architectures, are generally usable in a pipelined manner. This leads to the need for static scheduling of tasks on processor pipelines. This thesis considers problems associated with determining optimal schedules for such pipelines. A collection of algorithms is presented with their utility determined by the size and other characteristics of the system. The algorithms employ heuristics, dynamic programming and statistical methods to schedule tasks derived...

Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation

Datar, Seema, "Pipeline Task Scheduling with Appication to Network Processors" Report Number: WUCSE-2004-48 (2004). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1021

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1021

Pipeline Task Scheduling with Appication to Network Processors

Seema Datar

Complete Abstract:

Chip Multi-Processors (CMPs) are now available in a variety of systems and provide the opportunity for achieving high computational performance by exploiting application-level parallelism. In the communications environment, network processors (NPs), designed around CMP architectures, are generally usable in a pipelined manner. This leads to the need for static scheduling of tasks on processor pipelines. This thesis considers problems associated with determining optimal schedules for such pipelines. A collection of algorithms is presented with their utility determined by the size and other characteristics of the system. The algorithms employ heuristics, dynamic programming and statistical methods to schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Experimental results indicate that while the dynamic programming algorithm obtains the optimal schedules, heuristics and statistical methods obtain schedules within 10% of the optimal, 95% of the time. Examples are given to show the use of these algorithms for general pipeline/algorithm design and for use in the Network Processor environment with typical networking applications.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2004-48

Pipeline Task Scheduling with Appication to Network Processors, Master's Thesis, August 2004

Authors: Datar, Seema

August 4, 2004

Abstract: Chip Multi-Processors (CMPs) are now available in a variety of systems and provide the opportunity for achieving high computational performance by exploiting application-level parallelism. In the communications environment, network processors (NPs), designed around CMP architectures, are generally usable in a pipelined manner. This leads to the need for static scheduling of tasks on processor pipelines. This thesis considers problems associated with determining optimal schedules for such pipelines. A collection of algorithms is presented with their utility determined by the size and other characteristics of the system. The algorithms employ heuristics, dynamic programming and statistical methods to schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Experimental results indicate that while the dynamic programming algorithm obtains the optimal schedules, heuristics and statistical methods obtain schedules within 10% of the optimal, 95% of the time. Examples are given to show the use of these algorithms for general pipeline/algorithm design and for use in the Network Processor environment with typical networking applications

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

SEVER INSTITUTE OF TECHNOLOGY

MASTER OF SCIENCE DEGREE

THESIS ACCEPTANCE

(To be the first page of each copy of the thesis)

DATE: August 04, 2004

STUDENT'S NAME: Seema Datar

This student's thesis, entitled Pipeline Task Scheduling with Application to Network Processors has been examined by the undersigned committee of four faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Master of Science.

APPROVAL: ______________________________ Chairman

____

____

____

Short Title: Pipeline Task Scheduling Datar, M.Sc. 2004

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PIPELINE TASK SCHEDULING WITH APPLICATION TO NETWORK PROCESSORS

by

Seema Datar, B.E.

Prepared under the direction of Dr. Mark A. Franklin

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

August, 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

PIPELINE TASK SCHEDULING WITH APPLICATION TO NETWORK PROCESSORS

by Seema Datar

ADVISOR: Dr. Mark A. Franklin

August, 2004

Saint Louis, Missouri

Chip Multi-Processors (CMPs) are now available in a variety of systems and provide the opportunity for achieving high computational performance by exploiting application-level parallelism. In the communications environment, network processors (NPs), designed around CMP architectures, are generally usable in a pipelined manner. This leads to the need for static scheduling of tasks on processor pipelines. This thesis considers problems associated with determining optimal schedules for such pipelines. A collection of algorithms is presented with their utility determined by the size and other characteristics of the system. The algorithms employ heuristics, dynamic programming and statistical methods to schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Experimental results indicate that while the dynamic programming algorithm obtains the optimal schedules, heuristics and statistical methods obtain schedules within 10% of the optimal, 95% of the time. Examples are given to show the use of these algorithms for general pipeline/algorithm design and for use in the Network Processor environment with typical networking applications.

to my parents

Contents

List of Tables . . . . **vii**

List of Figures . . . . **viii**

Acknowledgments . . . . **x**

1 Introduction **1**

1.1 Generic NP Architecture 2

1.2 Problem Introduction 3

1.3 Related Work 5

1.4 Thesis Outline 7

2 Pipeline Task Scheduling Problem **9**

2.1 Problem Formulation 9

2.2 Assignment Constraints 11

2.3 Memory Contention Model 12

2.4 Performance Metrics 17

2.5 Complete Enumeration Method 18

3 Greedypipe - A Heuristic **21**

3.1 Greedypipe Algorithm 21

3.1.1 *Greedypipe*: Overall Algorithm 23

3.1.2 A Simple Example 24

3.2 Complexity of *Greedypipe* 26

3.2.1 Multiple Flows with no shared tasks 26

3.2.2 Multiple Flows with shared tasks 27

3.3 *GreedyPipe* Performance 27

4 Dynamic Programming . . . 31

4.1 Dynamic Programming for Task Scheduling . . . 32

4.1.1 Example . . . 35

4.1.2 Multiple Flows with Shared Tasks . . . 37

4.2 Complexity . . . 38

5 Simulated Annealing - A Statistical Approach . . . 40

5.1 Simulated Annealing for Task Scheduling . . . 42

5.1.1 Step Change . . . 42

5.1.2 Annealing Schedule . . . 43

5.1.3 Initial Task Assignment . . . 45

5.1.4 Multiple Flows . . . 45

5.2 Complexity . . . 46

5.3 Performance and Constraints . . . 47

6 Pipeline Design . . . 48

6.1 Pipeline Design using Scheduling Algorithms . . . 48

6.1.1 Number of Pipeline Stages and Number of Pipelines: . . . 50

6.1.2 Sharing of Tasks Between Flows . . . 51

6.1.3 Task Partitioning . . . 52

6.1.4 Number of Memory Banks and Memory Bank Assignment . . . 54

6.2 A Network Processor Problem . . . 55

6.2.1 Longest Prefix Matching (LPM) . . . 56

6.2.2 AES Encryption - A Pipelined Implementation . . . 59

6.2.3 Data Compression - A Pipelined Implementation . . . 61

6.2.4 NP Example Design Results . . . 62

6.2.5 Simulated Annealing for Task & Memory Assignment . . . 65

7 Comparative Study of Solution Approaches . . . 68

7.1 Complexity Analysis . . . 68

7.2 Execution Time and Accuracy . . . 70

7.2.1 SP/SF . . . 71

7.2.2 SP/MF . . . 72

7.2.3 SP/MF/ST . . . 73

7.2.4 SP/SF/SM . . . 75

7.2.5 SP/MF/SM . . . 76

8 Conclusions and Future Work . . . 78
8.1 Contributions . . . 78
8.2 Future Work . . . 79

Appendix A Memory Contention Delay . . . 81

Appendix B *Phoenix* Toolset Manual (ver 1.0) . . . 84
B.1 Usage . . . 84
B.1.1 Command Line Format . . . 85
B.1.2 Configuration File Format . . . 87
B.1.3 Output Format . . . 89
B.2 Example . . . 90

Appendix C *Phoenix* Toolset File Organisation . . . 94

References . . . 96

Vita . . . 101

List of Tables

2.1 A Single Flow, Single Shared Memory Example 13

3.1 A Single Flows with five *ordered* tasks 24

3.2 All possible allocations for Stage 1 24

3.3 Best allocations for Stage 2 25

3.4 Best allocations, forward direction 26

4.1 A Single Flow with five *ordered* tasks 35

4.2 Optimal Solutions at Stage 1 35

4.3 Possible States at Stage 2 36

4.4 Optimal State Values at Stage 2 36

4.5 Possible States at Stage 3 36

4.6 Optimal State Values at Stage 3 37

4.7 Optimal allocations 37

6.1 Task Times (μsec) for LPM, Encryption & Compression 59

6.2 Bandwidths for Best Assignments (Pps=Packets/second; Flow 1 packet length=40 Bytes; Flow 2 & 3 packet length=1500 Bytes) 64

6.3 Task Times (μsec) for AES Encryption assuming 15 clock cycles per memory access 65

7.1 Time Complexities 69

B.1 Sample - Interactive Command Line Option 84

B.2 Command Line Options For *Phoenix* 84

List of Figures

1.1 Generic Network Processor Architecture 2

1.2 NP Task Scheduling Problem Examples 4

2.1 Memory Contention Example 14

2.2 Worst Case Memory Contention Computation 16

2.3 Variation in execution time for flows with no shared tasks 19

2.4 Variation in execution time for flows with shared tasks 19

3.1 % Error vs % Cases with Error 27

3.2 % Error vs Number of Stages 28

3.3 % Error vs % Sharing 29

4.1 Dynamic Programming 33

5.1 Simulated Annealing Flow Diagram 41

5.2 A Step in Simulated Annealing 42

6.1 Throughput vs number of pipeline stages 50

6.2 Throughput vs number of pipelines vs number of stages 51

6.3 Throughput vs percent shared tasks 52

6.4 Throughput vs task partitioning 53

6.5 Throughput versus Number of Memory Banks 54

6.6 Example Longest Prefix Match Tree 58

6.7 Throughput vs. Number of Stages (A single separate pipeline for each application) 59

6.8 Rijndael Algorithm Implementation Block Diagram 61

6.9 Pipelined Implementation of the *LZW* Algorithm 62

6.10 Two Pipelines - Throughput vs. Num. Stages (X, Y –> X stages for Pipe 1 & Y stages for Pipe 2) 65

6.11 Three Pipelines - Throughput vs Num. stages for Pipelines 2 & 3; Pipeline 1=4 stages 65
6.12 Throughput vs Number Memory Banks 66

7.1 Execution Time, Single Flow 71
7.2 % Error in Throughput, Single Flow (SA is optimal over all experiments) 71
7.3 Execution Time for Multiple Flows. 72
7.4 % Error in Throughput, Multiple Flows. 72
7.5 Execution Time for Multiple Flows, Shared Tasks 73
7.6 % Error in Throughput, Multiple Flows, Shared Tasks 73
7.7 Execution Time; Single Flow, Memory Contention 75
7.8 % Error in Throughput, Single Flow, Memory Contention 75
7.9 Execution Time; Multiple Flows, Memory Contention 76
7.10 % Error in Throughput; Multiple Flows, Memory Contention 76

A.1 Multi-flow Memory Contention 82

B.1 Sample - *Phoenix* Toolset Configuration File 87
B.2 Sample - *Phoenix* Toolset Output 89
B.3 Sample - *Phoenix* Interactive Execution 91
B.4 Example - config.txt 92
B.5 Example - *Phoenix* Output 93

Acknowledgments

I would like to thank my advisor, Dr. Mark A. Franklin, for his supervision, guidance, insightful suggestions and support for the last two years. I would also like to thank the other members of my thesis defense committee, Dr. Roger Chamberlain, Dr. Patrick Crowley and Dr. Jason Fritts, who offered their time, experience and expertise in the evaluation of my research.

I would like to thank all my friends in the CSE department for the wonderful time spent at work. I would in particular like to thank Vinayak Joshi and Sarang Dharmapurikar for sharing their time and knowledge throughout the last two years. I would also like to thank Praveen Krishnamurthy, Roopa Pundaleeka and Bharath Madhusudan for the many enjoyable experiences we shared.

The research represented in this thesis was supported in part by NSF under grant CCR-0217334 and I would like to thank the organization for its financial support.

I would also like to acknowledge Jean Grothe, Myrna Harbison, Peggy Fuller, Sharon Matlock and Stella Sung for their efforts in making the CSE department run effectively.

Finally I would like to thank my parents for all their love and support.

Seema Datar

Washington University in Saint Louis
August 2004

Chapter 1

Introduction

Due to increased data rates and the requirements of more sophisticated networking protocols and functions, networking solutions continue to demand more powerful processing capabilities. These processing requirements are in flux and continue to change and evolve. Thus, another important aspect of satisfying processing needs is that the solution have the flexibility to respond to changing requirements. This has led to the development of the Network Processor (NP), a software programmable device with architecture and features, that are designed for efficient packet processing. It achieves a high programming power by employing multiple programmable processing engines (PEs), by adding special instructions targeted to networking applications, and by having dedicated on-chip logic implementing selected complex functions. Thus, the NP combines the best of both worlds; the flexibility/programmability of a generic processor and the speed of hardwired ASIC solutions. Network Processors (NPs), designed around Chip Multi-Processor (CMP) architectures, may be used in a pipelined manner. This leads to the issue of scheduling tasks on processor pipelines. The research considered in this thesis evaluates problems associated with determining optimal schedules for such pipelines.

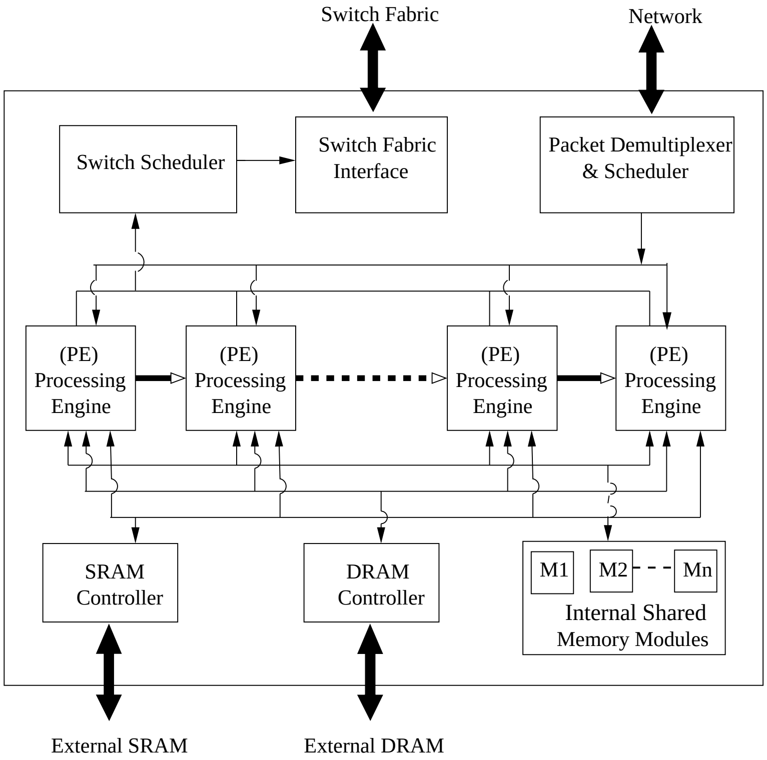


Figure 1.1: Generic Network Processor Architecture

1.1 Generic NP Architecture

Figure 1.1 shows the layout of a typical Network Processor. The PEs in an NP are typically arranged in a parallel or a pipelined fashion and NPs with as many as sixteen PEs are currently commercially available [20, 21, 27]. The arrangement is configurable and selecting the best one is important in achieving high performance. The data path of a packet begins at the switch fabric interface where the packet is received and passed on to the Packet De-multiplexer and Scheduler unit. The scheduler performs packet classification and allocates the packets to the appropriate PE resources. Note that the NP may be configured to have multiple PE pipelines and packets may require assignment to a particular pipeline depending on their processing requirements. After progressing through a pipeline of PEs, the packet is returned to the switch fabric through the switch scheduler and is transmitted to the next node of the network. NPs have a fast access on-chip RAM in addition to external DRAM (for large routing and classification tables) and SRAM capabilities. Additionally, the on-chip RAM (i.e., internal shared memory) may consist of multiple modules with one or more PEs being associated with a given module. Note that when memory modules are shared, contention for this resource can result in a loss of performance.

1.2 Problem Introduction

This research concentrates on NPs that are configured into one or more PE pipelines. We consider situations where packet applications are segmented into an ordered set of tasks with these tasks assigned to a pipeline of PEs (each PE is a pipeline stage). Thus the packet is processed in parts as it progresses through the pipeline, executing each task of the application in turn. Packet processing may involve multiple applications and different packet types may require different application sets for their processing. We refer to these application sets as *flows*. Certain applications could be common across application sets so that there may be shared tasks between flows. There may also be more than one pipeline available for allocation, so that an effective assignment of flows to pipelines would also need to be done.

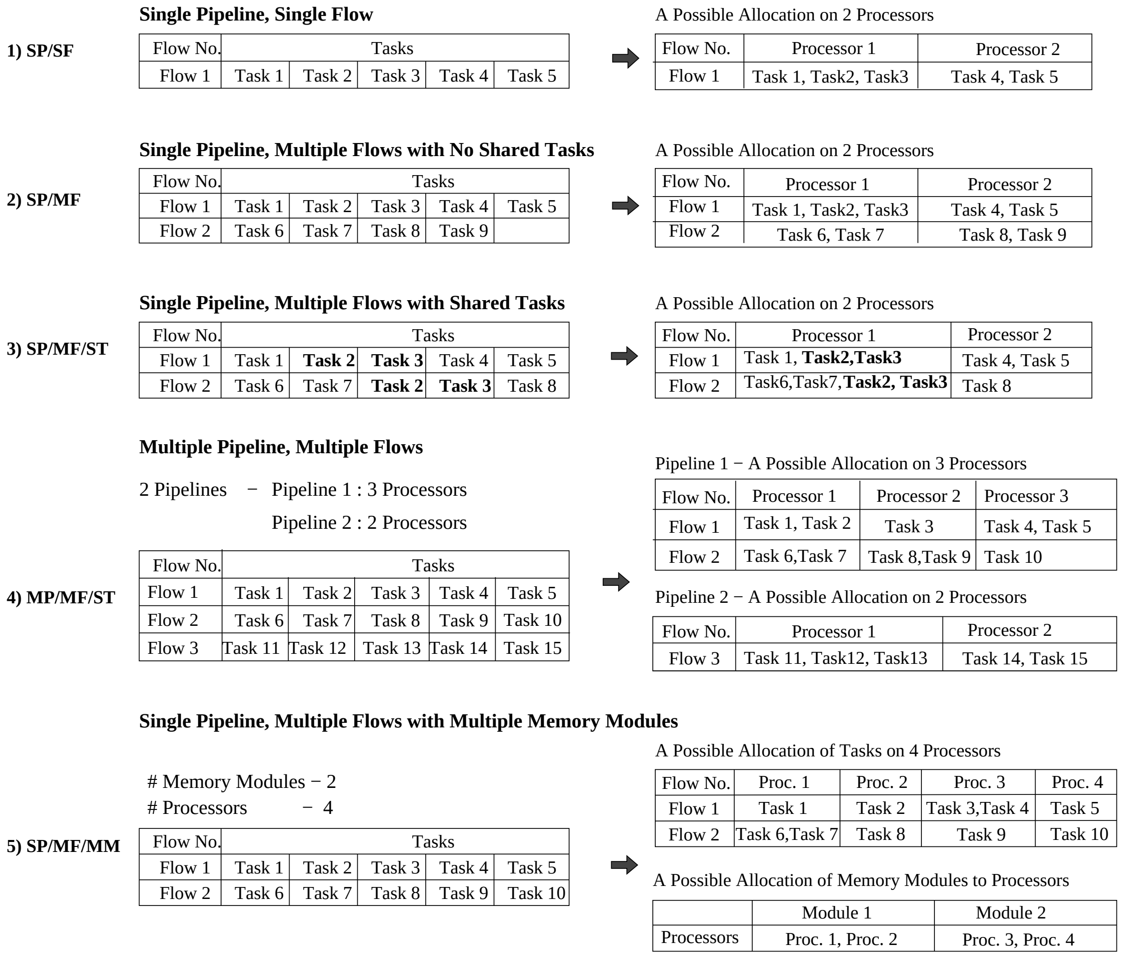
Considering all these factors, we can categorize the scheduling problem in order of increasing complexity as follows. Figure 1.2 illustrates the different scheduling problem types and provides sample task allocations.

1. **S**ingle **P**ipeline, **S**ingle **F**low (**SP/SF**) - Tasks in a single flow are allocated to a single pipeline of processors.

2. **S**ingle **P**ipeline, **M**ultiple **F**low (**SP/MF**) - Tasks of multiple flows need to be allocated to a single pipeline of processors.

3. **S**ingle **P**ipeline, **M**ultiple **F**low, **S**hared **T**asks (**SP/MF/ST**) - This is like SP/MF except that there may be shared tasks between the flows.

4. **M**ultiple **P**ipelines, **M**ultiple **F**lows (**MP/MF**) - More than one pipeline may be available so that while tasks need to be optimally allocated to processors,



1) SP/SF

Single Pipeline, Single Flow

Flow No.	Tasks				
Flow 1	Task 1	Task 2	Task 3	Task 4	Task 5

A Possible Allocation on 2 Processors

Flow No.	Processor 1	Processor 2
Flow 1	Task 1, Task2, Task3	Task 4, Task 5

2) SP/MF

Single Pipeline, Multiple Flows with No Shared Tasks

Flow No.	Tasks				
Flow 1	Task 1	Task 2	Task 3	Task 4	Task 5
Flow 2	Task 6	Task 7	Task 8	Task 9	

A Possible Allocation on 2 Processors

Flow No.	Processor 1	Processor 2
Flow 1	Task 1, Task2, Task3	Task 4, Task 5
Flow 2	Task 6, Task 7	Task 8, Task 9

3) SP/MF/ST

Single Pipeline, Multiple Flows with Shared Tasks

Flow No.	Tasks				
Flow 1	Task 1	**Task 2**	**Task 3**	Task 4	Task 5
Flow 2	Task 6	Task 7	**Task 2**	**Task 3**	Task 8

A Possible Allocation on 2 Processors

Flow No.	Processor 1	Processor 2
Flow 1	Task 1, **Task2,Task3**	Task 4, Task 5
Flow 2	Task6,Task7,**Task2, Task3**	Task 8

4) MP/MF/ST

Multiple Pipeline, Multiple Flows

2 Pipelines − Pipeline 1 : 3 Processors
Pipeline 2 : 2 Processors

Flow No.	Tasks				
Flow 1	Task 1	Task 2	Task 3	Task 4	Task 5
Flow 2	Task 6	Task 7	Task 8	Task 9	Task 10
Flow 3	Task 11	Task 12	Task 13	Task 14	Task 15

Pipeline 1 − A Possible Allocation on 3 Processors

Flow No.	Processor 1	Processor 2	Processor 3
Flow 1	Task 1, Task 2	Task 3	Task 4, Task 5
Flow 2	Task 6,Task 7	Task 8,Task 9	Task 10

Pipeline 2 − A Possible Allocation on 2 Processors

Flow No.	Processor 1	Processor 2
Flow 3	Task 11, Task12, Task13	Task 14, Task 15

5) SP/MF/MM

Single Pipeline, Multiple Flows with Multiple Memory Modules

# Memory Modules − 2
# Processors − 4

Flow No.	Tasks				
Flow 1	Task 1	Task 2	Task 3	Task 4	Task 5
Flow 2	Task 6	Task 7	Task 8	Task 9	Task 10

A Possible Allocation of Tasks on 4 Processors

Flow No.	Proc. 1	Proc. 2	Proc. 3	Proc. 4
Flow 1	Task 1	Task 2	Task 3,Task 4	Task 5
Flow 2	Task 6,Task 7	Task 8	Task 9	Task 10

A Possible Allocation of Memory Modules to Processors

	Module 1	Module 2
Processors	Proc. 1, Proc. 2	Proc. 3, Proc. 4

Figure 1.2: NP Task Scheduling Problem Examples

flows also need to be effectively mapped to pipelines. A given flow must be implemented entirely within a single pipeline.

5. **S**ingle **P**ipeline, **M**ultiple **F**lows, **M**ultiple **M**emory **M**odules(**SP/MF/MM**) - While there may be multiple modules of internal memory available, they may need to be shared between processors. This kind of sharing of resources could lead to contention for resources (memory in this case) and affect the performance. The task allocation in this case should be made with consideration to the memory resource contention. Extensions to this problem class include MP/MF/MM and MP/MF/ST/MM.

The line rates that can be supported by the system depend on the pipeline(s) throughput which, in turn, depends on a number of factors including just how application subtasks are assigned to the PEs. In our research, we mainly focus on maximizing the system throughput by effective allocation of tasks to processors. Three different approaches are used towards solving the task scheduling problem; a heuristic approach, dynamic programming and a statistical approach using *Simulated Annealing.* The subsequent chapters describe these approaches and the limitations of each one of them when applying them to the above described classes of problems.

Additionally, we also evaluate the usability of a tool, based on the above mentioned approaches, for designing Network Processors given that there is *a priori* knowledge of the flow types that will be serviced, the performance requirements, and the costs associated with implementation (e.g., chip area). In architectures where there are a limited number of on-chip RAM modules, assignment of these modules to one or more PEs affects performance. Other design issues relate to determining the number of stages a pipeline must have. While this is clearly related to the line rates one would like to achieve, and the complexity of the applications, there are a number of design options available, each of which requires that a good assignment of application tasks to pipeline stages be obtained. Another example is determining whether algorithmic efforts at changing the number of tasks or task durations associated with an application might help in improving system throughput. We demonstrate the effect of various elements of NP design on the performance of an NP by a range of experiments conducted using the scheduling tool-set developed as part of this thesis.

1.3 Related Work

There is a long history associated with related problems in deterministic job-shop scheduling [22] and these problems have been investigated from a variety of perspectives including integer programming, heuristics, and other approaches. Similar problems have also been dealt with in the context of finding compilation techniques for general purpose parallel languages on multiprocessors[39, 35]. The primary objective of the compilation techniques is to minimize the response time while simultaneously reducing overhead due to inter-process synchronization and communication over a general parallel processor. Multiprocessor performance in these cases is maximized by identifying potential parallelism and then partitioning the program accordingly to exploit the parallelism.

Scheduling of jobs to processors so that a given cost function is minimized is an important problem in many areas of computer science. In parallel computing, with no special constraints on the jobs or the processors, minimizing the total completion time has been proved to be NP-Hard [14], even for two processors. Thus a number of restricted versions of the scheduling problem have been investigated in order to make it tractable. One such class of problems, referred to as Structured Data Partitioning problems or MinMax problems, impose an implicit order on a sequence of n elements. The objective is to partition the elements into a sequence of p intervals such that the maximal value of the cost function, of each interval is minimized (hence the name MinMax). The first reference to the MinMax problem was made by A.Bokhari for single dimensional partitioning problems in parallel, pipelined and distributed computing in [36] where he presented an $O(n^3p)$ algorithm using a bottleneck-path algorithm. Anily and Federgruen [33] and Hansen and Lih [18] independently presented the same dynamic programming algorithm reducing the time complexity of the solution to $O(n^2p)$. Manne and Sorevik then presented, in [26], an $O(p(n-p)\log p)$ algorithm based on iteratively improving a given partition. They also described a bisection method for finding an approximate solution which runs in time $O(n \log(f(0, n-1/e))$, where e is the desired precision. Further improvements to the iterative approach were suggested by Pinar and Aykanat in [31]. They use improved algorithm initialization methods. Olstad and Manne studied the problem in the context of issues related to load balancing when performing sparse matrix computations on parallel computers. They refined the dynamic programming approach to a complexity of $O(p(n-p))$ and applied it for partitioning of acyclic graphs too.

There exists a rich literature on scheduling of tasks to processors and an overview of it can be found in [16]. The MinMax problem can be generalized to dimensions higher than one however, all the cited solutions are pertinent to single dimensional partitioning problems.

Real-time packet scheduling problems have also been considered in the context of network processors [43]. In this case, however, packets were assumed to be completely processed on a single processor. A primary concern in that work was to assign packets to processors in a manner that minimized the effect of cold cache misses on performance.

Our work aims at maximizing the throughput of a pipelined CMP by effective assignment of flow tasks to pipeline stages. It differs from the prior work cited in a number of ways. Primarily, the problem definition differs from those considered in the past in that we consider multiple flows and pipelines, sharing of tasks on pipeline stages, sharing of resources (e.g., memory) between stages and employing a bandwidth performance metric associated with the requirements of the computer pipeline environment. Thus, while limited problem classes (e.g., 1 - SP/SF) can be solved using some of the existing techniques, the more complex classes noted above have required consideration of new approaches.

1.4 Thesis Outline

Chapter 2 presents a model for the assignment of application tasks and memory modules to pipeline stages. Given this formulation, Chapter 3, 4 and 5 indicate how heuristics, dynamic programming and a statistical optimization method (*Simulated Annealing*) can be used to obtain the optimal task and memory assignment. Chapter 6 then illustrates the use of this capability first in a set of synthetic problems, and then by applying it to a specific NP problem where the applications include routing, compression and encryption. Chapter 7 compares the three algorithmic approaches for the task scheduling problem and concludes with a summary and a discussion of future work.

Chapter 2

Pipeline Task Scheduling Problem

2.1 Problem Formulation

In this chapter, we give a formal definition of the task scheduling problem and lay down the constraints associated with it. We also describe a resource (memory in this case) contention model and use it to formulate the performance metrics for the scheduling problem. Network processors (NPs) typically have multiple input flows where, for our research, we define a flow as a set of successive functions that must be performed on packets belonging to the flow[1]. For example, one flow may require that incoming packets be compressed (say using Lempel Ziv) and then routed using a specific LPM (Longest Prefix Match) algorithm. Another flow might require packet encryption, transcoding and then routing. We consider here application algorithms that may be pipelined and are implemented on a pipeline of identical processors. The general issue of how to develop a pipelined algorithm for a given application is not considered here except for the special cases of Longest Prefix Match (LPM), encryption (AES) and compression (LZW) that are discussed in Chapter 7. Each processor in the pipeline operates on a packet, does some partial processing associated with the application, and then passes the packet (generally modified) along with other

[1] Flows correspond to a sequence of functions rather than packets associated with source-destination pairs.

information to the next processor in the pipeline[2]. After passing through the pipeline, the packet is sent into a switch and from there into the network. We assume that there is a steady stream of packets arriving.

Each packet belongs to one flow and flows are represented by the set F:

$$F = \{F_1,F_2,F_3, \ldots ,F_N\}$$

The processing associated with flow applications can be partitioned into an *ordered* set of tasks, $Tasks_j$. Each task in $Tasks_j$ is represented by T_{ij}, where i $(1 \leq i \leq M_j)$, and j $(1 \leq j \leq N)$ respectively designate the task and flow number. Thus $Tasks_j$ is the set of tasks associated with flow j and M_j is the total number of tasks for flow j given by:

$$Tasks_j = \{T_{1j},\, T_{2j},\, T_{3j},\, \ldots ,T_{M_jj}\}$$

Corresponding to the tasks are task times (i.e., the time for executing the tasks on a given pipeline stage), t_{ij}. For flow j, the set of task execution times is:

$$TaskTimes_j = \{t_{1j},\, t_{2j},\, \ldots ,t_{M_jj}\}$$

These times will be obtained experimentally using a processor simulator under zero cache miss conditions. This permits us to separate out the effects of memory architecture from the pure computational requirements of the applications. To take application/task memory usage into account, each task has associated with it the number of memory accesses that occur during its execution, m_{ij}. Each PE in an NP is assumed to have local instruction memory and the memory accesses are primarily required for data accesses. the memory accesses will also be obtained through simulations. Thus, for flow j:

$$MEM_j = \{m_{1j},\, m_{2j},\, \ldots ,m_{M_jj}\}.$$

The resources associated with the NP consist of PEs that may be arranged in a pipeline manner (K identical processor stages for a single pipeline) and memory modules (Q modules) with $Q \leq K$. Thus, the set of PEs, P, and memory modules, M, are given by :

[2] Note that there is generally an overhead associated with moving data between successive stages. This can be easily dealt within the framework provided. However, for a constant overhead between stages, this typically affects the pipeline latency but not throughput and thus does not impact task scheduling

$$P = \{P_1,\, P_2,\, P_3,\; \dots\, , P_K\}$$
$$M = \{M_1,\, M_2,\, M_3,\; \dots\, , M_Q\}$$

The memory modules may be on-chip or off-chip and may be shared between the PEs. The K PEs can also arranged into multiple pipelines. A two pipeline case would be represented as :

$$Pipeline^1 = \{P_1,\, P_2,\, P_3,\; \dots\, , P_l\}$$
$$Pipeline^2 = \{P_{l+1},\, P_{l+2},\, P_{l+3},\; \dots\, , P_K\}$$

Initially we present a less complex model, where memory delays are not a significant factor and may be ignored in the task assignment process. Later when memory issues are considered, the association of a processor k with a memory block m is represented by the variable A_{km}. $A_{km} = 1$ indicates that the processor k has access to the memory block m, while $A_{km} = 0$ indicates otherwise. If $Q < K$, then multiple processors access the same memory block, leading to contention amongst the processors for memory access. This may have a significant effect on system performance and is considered later in the discussion of performance metrics.

2.2 Assignment Constraints

The task assignment problem consists of mapping the full set of tasks onto the stages of a pipeline in a manner that preserves task ordering within a flow and optimizes a given performance metric. The assignment of task i from flow j to processor stage k can be expressed using the binary variable X_{ijk} where $X_{ijk} = 1$ if the task is assigned to the processor, and $X_{ijk} = 0$ otherwise. Thus, the number of tasks on a processor k is given by:

$$P_{num.k} = \sum_{j=1}^{N} \sum_{i=1}^{M_j} X_{ijk} \tag{2.1}$$

Additionally, the following three constraints apply:

- The assignment process must maintain sequential task ordering. Thus, for l, $1 \leq l \leq M_j$, for all i, j, k, r if $X_{ijk} = 1$ and $X_{(i+l)jr} = 1$ then $k \leq r$.

- A task may only be assigned to a single processor. Thus for a given task i from flow j, $\sum_{k=1}^{R} X_{ijk} = 1$.

- An additional constraint is applied in situations where the same task is associated with multiple flows. Designating tasks to be shared across flows implies that there will be a single instantiation of the shared task and it will be assigned to a single pipeline stage. Thus, for the case of two flows, j, s, and two tasks, i, r, that are the same and are to share the same stage (and code):

  $$\text{if } T_{ij} = T_{rs} \text{ and } X_{ijk} = 1 \text{ and } X_{rsm} = 1 \text{ then } k = m.$$

  This can be extended naturally to more than two flows. If it is not desired to have such sharing even though the tasks are the same, this can be dealt with by giving the tasks different names.

The third constraint helps in conserving code space and instruction cache misses since the application code does not get duplicated. Additionally, in designs employing caches, it can help to reduce cache misses. However this constrains how tasks can be assigned and thus, in certain cases may lead to a loss in performance. The general problem considered therefore is just how to assign the tasks from each flow to the pipeline(s) stages, in a manner that satisfies the above constraints while maximizing an appropriate performance metric. Performance metrics are considered in Section 2.4. The next section presents a model for memory contention in the task scheduling problem.

2.3 Memory Contention Model

The assignment of memory modules to pipeline stages may strongly impact system performance. Since each of the pipeline stages that accesses a shared module may encounter a delay due to contention, any task and memory module assignment must include a method for evaluating these contention delays. If, ideally, a very fast simulation of the entire NP along with the flows and associated applications were available, then successive simulation runs could provide performance information from which the best assignment could be derived. This, however, is not practical since such simulations take a long time, and to find the best assignments requires a large number of simulation executions.

Table 2.1: A Single Flow, Single Shared Memory Example

$Flow$ Tasks	T_1	T_2	T_3
$Flow$ Task Times	t_1	t_2	t_3
$Flow$ Memory Accesses	w_{11},w_{12}	w_{21},w_{22},w_{23}	w_{31}
Pipeline	P_1	P_2	P_3

In our research, we adopt a *worst case* memory contention model. This permits simple evaluation of memory contention delays without having to resort to a full NP simulation. It also corresponds to the general requirement that networking components must meet line rates under worst case conditions and, since processors and memory accesses across pipeline stages are not synchronized in time, such worst case conditions may occur. We are concerned here principally with data memory accesses, and assume that sufficient local instruction memory is present on each processor stage so that contention delays are not encountered when accessing instructions. The technique employed may be best explained by use of a simple example where a single memory module is being considered. Assume that the memory module is shared by three processors in a single pipeline and single flow F_1 is present consisting of three consecutive tasks T_1, T_2 and T_3. The tasks are assigned to the processors P_1, P_2 and P_3 in the three-stage pipeline. Associated with each task is a set of successive memory accesses that are designated by w_{ij} where i is the task number and j is the memory access number. Thus, w_{12} is the second memory access for task 1. These are shown for this system in Table 2.1.

Figure 2.1 shows two possible rounds of data memory accesses for the processors and depicts a worst case scenario for the memory accesses made by P_1. In the first round, while the memory request by P_3 (w_{31}) is being processed the other two processors (P_2 and P_1) also make their first memory requests (w_{21} and w_{11}) and these

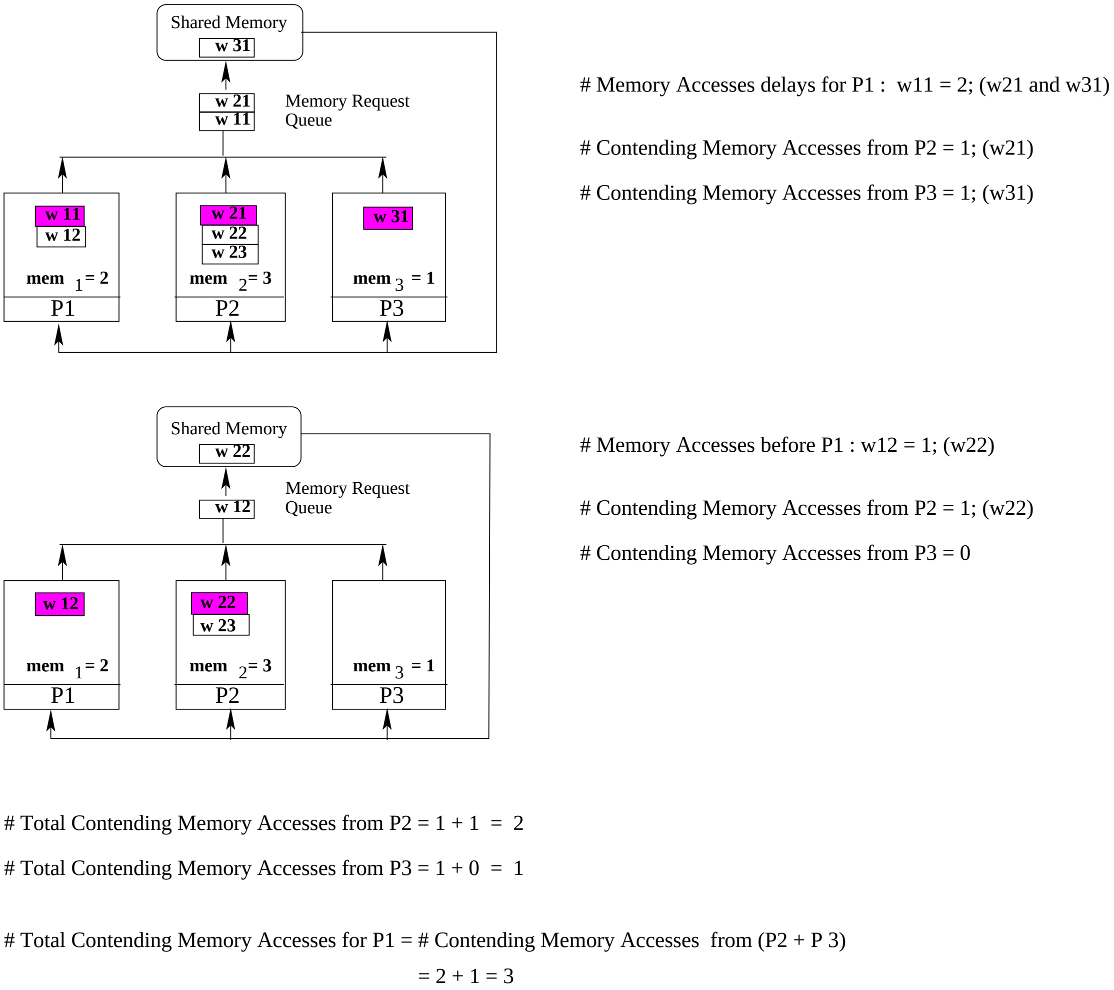

Shared Memory
w 31
Memory Request Queue
w 21
w 11
w 11
w 12
w 21
w 22
w 23
w 31
mem 1 = 2
mem 2 = 3
mem 3 = 1
P1
P2
P3
# Memory Accesses delays for P1 : w11 = 2; (w21 and w31)
# Contending Memory Accesses from P2 = 1; (w21)
# Contending Memory Accesses from P3 = 1; (w31)
Shared Memory
w 22
Memory Request Queue
w 12
w 12
w 22
w 23
mem 1 = 2
mem 2 = 3
mem 3 = 1
P1
P2
P3
# Memory Accesses before P1 : w12 = 1; (w22)
# Contending Memory Accesses from P2 = 1; (w22)
# Contending Memory Accesses from P3 = 0
# Total Contending Memory Accesses from P2 = 1 + 1 = 2
# Total Contending Memory Accesses from P3 = 1 + 0 = 1
# Total Contending Memory Accesses for P1 = # Contending Memory Accesses from (P2 + P 3)
= 2 + 1 = 3

Figure 2.1: Memory Contention Example

requests must now wait. So, while w_{31} is being serviced, requests w_{21} and w_{11} get queued up and serviced in a FIFO fashion. In this situation w_{11} must wait until the two requests (w_{31},w_{21}) before it are completed, and only after that is w_{11} processed. Similarly, if there were different orderings of requests , P_2's and P_3's requests might also encounter such delays. Given the number of memory requests associated with each task and a given task to stage assignment, the worst case memory delay for a stage can be evaluated. Once this worst case delay is known, the worst case task execution time, W, can be evaluated in terms of two components, the task execution time without data memory delays and the worst case memory waiting time. Thus:

$$W = \textit{Task Execution time} + \textit{Memory Waiting Time} \tag{2.2}$$

We assume that after a memory request has been made, a single threaded processor must stall until the request is satisfied. Therefore, there cannot be more than one pending request from a processor. Thus, the maximum number of memory requests at any point in time will be equal to the number of processors (given that each processor has a memory request) accessing the memory block.

While the example above has a single memory module, in general there may be multiple modules available and part of the assignment task is to assign memory modules to processor stages. Following the example above, for a given memory block m, the maximum memory contention/waiting time for any processor is equal to *(*$R_m - 1$*)* $\times$ t_{mem} where R_m is the number of processors assigned to the memory block m and t_{mem} is the time required to process a single memory request with no contention. Since in the considered example there is only a single memory module shared between processors, we can drop the subscript m for this case. In this case all the processors are contending, but contention is generally less than *(*$R_m - 1$*)* $\times$ t_{mem} since the number of memory requests associated with each stage is different and, after a few requests have been satisfied, some processors will not have any requests remaining while others may. This is illustrated by a possible second round of memory accesses shown in Figure 2.1. The number of memory requests required by the processors is represented by mem_1, mem_2 and mem_3. In the first round of memory requests, the waiting time for w_{11} is *(*$R-1$*)* $\times$ t_{mem} with $R = 3$, since all the three processors have made a memory request (*the worst case*). However, in the second round of memory accesses, again assuming a worst case so that all the processors make memory requests together, since processor P_3 does not have any more memory accesses, the memory request by processor P_1 (w_{12}) faces contention only due to the memory request by processor P_2 (w_{21}). Note that, after the second round, P_1 is finished with all it's memory requests and thus, does not face any more contention from the remaining memory requests (w_{23}).

```
Total_Rounds = $mem_k$
round = 1
while (round < Total_Rounds)
Begin
    Max_Contention_For_Round = R
    for ( j = 1 to R )
    Begin
        if  ( $mem_j$  ==  0 ) AND ( j != k)
            Max_Contention_For_Round = Max_Contention_For_Round − 1
        else
            $mem_j$  = $mem_j$  − 1
    End
    Total_Contention = Total_Contention + Max_Contention_For_Round
    round = round + 1
End
Total_Waiting_Time = Total_Contention * $t_{mem}$
```

Figure 2.2: Worst Case Memory Contention Computation

Thus, given the number of memory requests associated with each task and the task to stage assignment, the memory contention can be computed using the algorithm shown in Figure 2.2. In the figure, mem_k refers to the memory accesses required by processor k. As shown, the total number of rounds (*Total_Rounds*) of memory access is equal to the number of memory accesses for the processor, k, for which the maximum contention needs to be obtained. At each round the number of contending requests (*Maximum_Contention_For_Round*) is initialized to R (the total number of processors) assuming that all the processors need to make memory requests. Subsequently, the number of remaining memory requests for each processor is checked and if a processor has completed all it's memory requests, *Maximum_Contention_For_Round* is reduced by one since that processor does not contend for memory any more. After each round, the number of memory requests required by each processor is reduced by one to account for the memory request completed in that round for each processor. The *Maximum_Contention_For_Round* for all the rounds is added up to obtain the total waiting time for a processor to complete all it's memory requests.

The expression to calculate the worst case memory delay for each processor, for the above described memory contention model is presented in Appendix A.

2.4 Performance Metrics

Given the formation above, to determine an optimal assignment it is necessary to specify a performance metric. The metric of interest in the NP environment generally relates to maximizing pipeline throughput (i.e., the number of packets processed per second).

Consider the case where there are one or more flows, flows that may share tasks, and (for simplicity) a single pipeline. Assume that pipeline throughput is limited by the maximum stage execution time taken over all stages in the pipeline. The execution time for a stage is determined by the tasks assigned to each stage and the memory modules assigned to each processor. Thus, the execution time for a single flow j, on a given stage k is given by:

$$s_{jk} = \sum_{i=1}^{M_j} X_{ijk} t_{ij} + (c_{jk} * t_{mem}) \tag{2.3}$$

where $(c_{jk} * t_{mem})$ is the additional time required due to contention for memory. c_{jk} as described in the previous section, represents the number of contending memory accesses for tasks of flow j that are assigned to stage k and t_{mem} is the time required to process a single memory request without contention. The worst case value of c_{jk} can be determined as described in the previous section. The maximum stage execution time for flow j across all the R stages is:

$$P_j = \max_{k=1}^{R}[s_{jk}] \;=\; \max_{k=1}^{R}\left\{\sum_{i=1}^{M_j} X_{ijk} t_{ij} + (c_{jk} * t_{mem})\right\} \tag{2.4}$$

The maximum stage execution time over all flows and stages is given by:

$$P = \max_{j=1}^{N} P_j = \max_{j=1}^{N}\left\{\max_{k=1}^{R}\left\{\sum_{i=1}^{M_j} X_{ijk} t_{ij} + (c_{jk} * t_{mem})\right\}\right\} \tag{2.5}$$

To maximize packet throughput, the problem becomes one of finding a task and memory assignment that minimizes P since *packet throughput* $\approx 1/P$.

2.5 Complete Enumeration Method

A straight forward method of finding the task and memory assignment that minimizes P is to perform a complete enumeration of all possible assignments, identify feasible assignments (i.e., those that satisfy the constraints discussed above), and select the optimum one. Since the multi-processor pipelines are supposed to be programmed offline, this method may not seem very expensive. However, even restricted versions of this problem have been proven to be NP Hard [22] and for a complex configuration of flows, stages and tasks per flow, it could take years to compute the optimal assignment, using the complete enumeration method. However, the complete enumeration method was implemented and used in evaluation of the other approaches for limited cases when the problem size was small.

Consider first the complexity associated with performing only task assignments for the single flow case where there are M tasks present and R identical pipeline stages. An upper bound on the number of combinations that must be examined for each flow, corresponds to the standard problem of calculating the number of ways of placing M identical balls into $(M + R - 1)$ bins and is given by :

$$Number\ of\ Assignments\ For\ Each\ Flow \leq \binom{M_j + R - 1}{M_j} \tag{2.6}$$

With N flows present each flow having M_j tasks, the number of combinations increases by the product of combinations possible with each flow. Thus:

$$Number\ of\ Assignments\ \leq\ \prod_{j=1}^{N} \binom{M_j + R - 1}{M_j} \tag{2.7}$$

Accordingly, assuming that the complexity of obtaining each assignment is $O(1)$, the time complexity of enumerating all the task assignments will be approximately $O((\frac{(M+R)!}{M!R!})^N)$. However if the flows are independent with no shared tasks and a separate memory module for each processor stage, the combinations for each flow can be enumerated independently and the complexity in that case will be given by

$O((\frac{(M+R)!}{M!R!})N)$. Equation 2.7 results in the worst case count of combinations. This will decrease (sometimes significantly) when there are more tasks common between the flows (i.e., shared), however, even with this reduction, the factorial and product components may result in the complete enumeration approach being infeasible for analyzing practical systems. The complexity increases further if the memory modules are shared between stages. With shared memory modules, the contention for each of the R stages needs to be computed. The complexity for computation of contention for each stage will depend on the memory module sharing. Thus, considering a worst case scenario such that a single memory module is shared between all the R stages, from Equation A.3 (Appendix A), each memory contention computation for a processor requires a comparison of memory accesses with all the other $R-1$ processors. Given that, the complexity for memory contention calculation for all the R stages is approximately $O(R^2)$. Thus, the complexity of enumerating all the task assignments with memory contention calculation with a single memory module shared between all the R processors is given by $O((\frac{(M+R)!}{M!R!})^N \times (R^2))$.

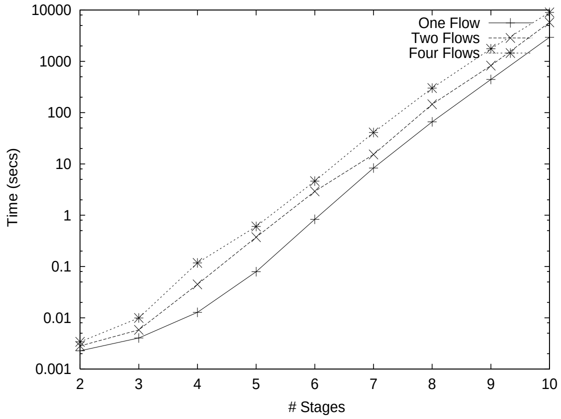


Figure 2.3: Variation in execution time for flows with no shared tasks

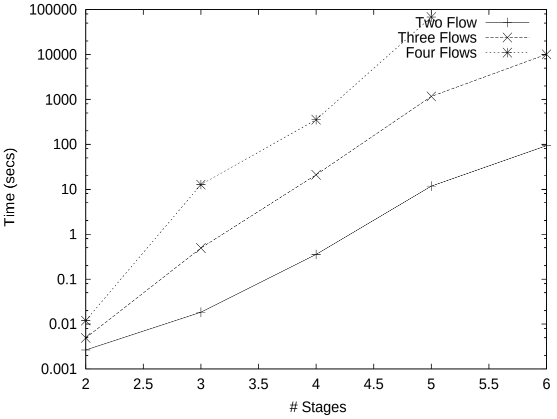


Figure 2.4: Variation in execution time for flows with shared tasks

Figures 2.3 and 2.4 illustrate the effect on the execution time of the complete enumeration method when the number of stages in a pipelined processor system is varied. Each datapoint represents an average of 10 experiments. The number of tasks in each flow is equal to twice the number of stages for each experiment.

Figure 2.3 shows experiments for systems with independent flows (no shared tasks) while Figure 2.4 shows experiments for flows with shared tasks. With the increase in number of flows in the system, for systems with independent flows, there is a linear increase in execution time whereas for systems with shared tasks the increase in execution time is almost exponential. The figure with shared tasks shows limited experiments due to very high execution times for systems with more than 4 stages.

This chapter provided a formal definition of the task scheduling problem and the constraints associated with it. Additionally, it also presented a memory contention model for the class of problems with limited or shared resources (memory in this case). This model was subsequently used to present the performance metric for the task scheduling problem. Chapters 3, 4, and 5 introduce three different approaches to allocate tasks to processors and use the described metric to demonstrate the effectiveness of the approaches under the given constraints.

Chapter 3

Greedypipe - A Heuristic

Greedypipe is a heuristic based, in part, on a greedy algorithm. It gives no guarantee of finding an optimal solution, however, it provides solutions quickly and tests indicate that it finds a near optimal solution most of the time. Note also that the algorithm was developed for solving the cases of multiple flows with shared tasks, but does not directly extend to the shared memory environment. Section 3.1 describes the *Greedypipe* algorithm and uses an example to further illustrate the application of the algorithm. Section 3.2 discusses the complexity of *Greedypipe* and Section 3.3 the algorithm's performance.

3.1 Greedypipe Algorithm

Ideally, one would like an assignment where, for each flow, the total execution times of flow tasks associated with each stage are equal. Using the notation introduced in Chapter 2, the total time for executing the tasks associated with flow j is given by:

$$TotalTime_j = \sum_{i=1}^{M_j} t_{ij} \tag{3.1}$$

With R stages in the pipeline, as indicated, an optimal allocation of tasks to pipeline stages is one where the execution time for each stage is equal. Under these conditions, the ideal delay per stage for flow j is :

$$Ideal.Delay.per.Stage_j \;=\; TotalTime_j/R \tag{3.2}$$

and the resulting maximized throughput is :

$$Packet.Throughput_j = 1/Ideal.Delay.per.Stage_j \tag{3.3}$$

Thus, *Step 1* of the GreedyPipe is to calculate this ideal delay (Equation 3.2). Actual task times and assignments that satisfy the constraints noted in Chapter 1 will however generally result in unequal execution times associated at each stage. The best of the possible assignments, however, will be the one(s) that come closest to that ideal.

Consider the time for execution of all flow j tasks on stage k as given by:

$$t_{jk} = \sum_{i=1}^{M_j} X_{ijk} t_{ij} \tag{3.4}$$

Since, throughput is calculated from the inverse of the maximum stage execution time, the optimum assignment for flow j is one that minimizes the value of Var_j in the expression given by Equation 3.5.

$$Var_j = \max_{k=1}^{R} \{|[t_{jk}] \;-\; Ideal.Delay.per.Stage_j|\} \tag{3.5}$$

When multiple flows are present there are potentially shared tasks that complicates task assignment. However, various assignments will meet the above constraints and selecting the optimal now requires Equation 3.5 to be expanded so that the throughput across all the flows is maximized. This can be achieved by selecting the task to stage assignment that minimizes the maximum Var_j across all flows:

$$Var = \max_{j=1}^{N} Var_j = \max_{j=1}^{N} \;\left\{ \max_{k=1}^{R} \{|[t_{jk}] \;-\; Ideal.Delay.per.Stage_j|\} \right\} \tag{3.6}$$

This metric attempts to equalize both the distribution of tasks to stages on a per flow basis and also on an aggregate flow basis. Note that, at a given stage m, potentially there may be multiple allocations for which the minimized Var has the same value. A simple tie breaking algorithm is used that selects the assignment, over all flows, that minimizes the sum, S (Equation 3.7), of the differences between the ideal delays and the assigned delays.

$$S = \sum_{j=1}^{N} (\ |[t_{jm}] \ - \ Ideal.Delay.per.Stage_j| \ ) \tag{3.7}$$

3.1.1 *Greedypipe*: Overall Algorithm

The overall heuristic begins by calculating the *Ideal.Delay.per.Stage* for each of the flows (*Step 1*). Task to stage allocations start with the first processor stage. Two sets of tasks, satisfying the constraints, are selected from each of the flows for allocation to this processor (*Step 2*). The first set is chosen so that the variation, Var_j, given by Equation 3.5 is minimized and is also a positive number. The second set is chosen similarly, however, Var_j is required to be a negative number. Thus the ideal delay value is bracketed. Additional sets that satisfy the constraints may be chosen at the cost of increased complexity and execution time. At this point, there are two allocations associated with each flow for the first processor and thus there are thus $(2)^N$ possible combinations of flow allocations. Each of these combinations is examined and the "best" two are kept for use in performing task assignments for the next pipeline stage(*Step 3*). The best two correspond to the two that, for this stage, minimize Var as expressed in Equation 3.6 with $R = 1$.

Assignments for the next pipeline stage are now considered. The process begins by first calculating new $Ideal.delay.per.stage$ values based on *unallocated* tasks and the number of remaining pipeline stages. Next, each of the two best allocations from the prior stage is used as a starting point for determining the best task-to-stage assignments for the current stage. For each of these and for each flow, two "best" assignments (positive and negative) are selected. As before, all combinations of these flow assignments are then examined and the two that minimize Var (Equation 3.6) with $R = 2$ are now kept as starting points for considering the next pipeline stage (Stage 3). This process continues until all stages in the pipeline are examined and a complete assignment has been done. The best of the final stage two assignments is now kept.

Notice that the algorithm has an implicit ordering aspect to it such that tasks and stages are considered in their first-to-last order. While in general this does well, given that local conditions determine allocations at each stage, it will not always result in the optimal allocation. To improve the results one can apply the same heuristic, however, start from the last task and stage, and apply the heuristic in a last-to-first order. Thus, in *GreedyPipe* the algorithm is applied in both directions with the final assignment being the best of the two.

3.1.2 A Simple Example

To illustrate operation of the *GreedyPipe* we present a simple single flow example that has five tasks and a three stage pipeline. The task times are given in Table 4.1.

Table 3.1: A Single Flows with five *ordered* tasks

	Task 1	Task 2	Task 3	Task 4	Task 5
Flow 1	T_1	T_2	T_3	T_4	T_5
Task Execution Times	2	5	3	1	3

Begin by calculating the $Ideal.Delay.per.Stage$ as 4.66 (e.g., $(2+5+3+1+3)/3 = 4.66$). Next consider allocations to stage 1 and calculate Var_1. This is shown in complete form in Table 3.2 although clearly not all calculations need be done.

Table 3.2: All possible allocations for Stage 1

Stage 1 Allocation	Var_1	Best Selections
T_1	$Var_1 \;=\; 4.66 - 2 \;=\; 2.66$	+
T_1, T_2	$Var_1 \;=\; 4.66 - (2+4) \;=\; -1.34$	−
T_1, T_2, T_3	$Var_1 \;=\; 4.66 - (2+4+3) \;=\; -4.34$	
T_1, T_2, T_3, T_4	$Var_1 \;=\; 4.66 - (2+4+3+1) \;=\; -5.34$	
T_1, T_2, T_3, T_4, T_5	$Var_1 \;=\; 4.66 - (2+4+3+1+3) \;=\; -8.34$	

The best positive selection corresponds to a stage 1 assignment of T_1, while the best negative selection corresponds to T_1, T_2. Starting with these selections, stage 2 assignments are now considered. Two new $Ideal.Delay.per.Stage$ values are now calculated based on stage selections 1 and 2 above. They are:

$$Ideal.Delay.per.Stage_1 = [(5 + 3 + 1 + 3)/2] = 6.0$$

and

$$Ideal.Delay.per.Stage_2 = [(3 + 1 + 3)/2] = 3.5.$$

A new set of possible allocations is now calculated. Table 3.3 shows the four assignments associated (Equation 3.5) with the starting allocations of stage 1. From these four, the two best are selected for consideration (noted with a *). The heuristic then breaks the 2.3 tie by employing the tie breaking algorithm. From Equation 3.7, since 1.0 < 2.0, the second entry in the table is selected. Since *GreedyPipe* retains two choices at each stage, the lower two choices are now examined. They are tied both in terms of the Max calculation (1.7) and the use of tie breaker. Thus, for this situation one of them is arbitrarily selected.

Table 3.3: Best allocations for Stage 2

Stage 2 Allocation				
Stage 1	Stage 2	Var_2	Max $[\lvert Var_1\rvert, \lvert Var_2\rvert]$	Best Selections
T_1	T_2	$Var_2 = 6.0 - 4 = 2.0$	2.3	
T_1	T_2, T_3	$Var_2 = 6.0 - (4 + 3) = -1.0$	2.3	$*$
T_1, T_2	T_3	$Var_2 = 3.5 - 3 = 0.5$	1.7	
T_1, T_2	T_3, T_4	$Var_2 = 3.5 - (3 + 1) = -0.5$	1.7	$*$

The two stage 2 allocation choices are now the starting point for examining the possible assignments for stage 3 as shown in Table 3.4. Since this is the last stage, all the remaining assignments are dictated by prior stage assignments. For this example, as Table 3.4 indicates, there are two best assignments. The same process is now repeated starting from stage 3 and task 5 and working in the reverse direction. For this case, the best assignments are the same as obtained in the forward direction.

Additionally, the maximum stage delay (7) for these is also the same as obtained when using a complete enumeration approach to find the optimal. Thus, in this example the heuristic yielded the optimal assignment.

Table 3.4: Best allocations, forward direction

Stage 3 Allocation			
Stage 1	Stage 2	Stage 3	Max. Stage Time
$T_1, \quad T = 2$	$T_2, \quad T = 5$	$T_3, T_4, T_5, \quad T = 7$	7
$T_1, T_2, \quad T = 7$	$T_3, T_4, \quad T = 4$	$T_5, \quad T = 3$	7 *

3.2 Complexity of ***Greedypipe***

We now determine the time complexity of the *Greedypipe* algorithm. Since the algorithm execution varies depending on whether there are shared tasks between flows, we will find the complexity for two cases:

- Multiple flows with no shared tasks.
- Multiple flows with shared tasks.

3.2.1 Multiple Flows with no shared tasks

In this case, since there is no dependency between flows, each flow can be handled independently. We have seen that *Greedypipe* progresses from the first stage to the last stage and then for better results does a reverse traversal. The operations at each stage are almost the same. Thus, with no dependency between the flows, the allocation can be done for one flow at a time. Since the operations at each stage are almost the same, we will find the complexity for a single stage execution and extend it to include multiple stages and flows. At each stage, except for the first stage, there are two initial starting points and for each one, selections are made in accordance with Equations 3.5 and 3.6. The complexity of these operations per stage is roughly $O(1)$. Now, since the same operation needs to be done for all the stages and flows, the complexity for a complete execution of the algorithm would be given by $O(NR)$, where N is the number of flows and R is the number of stages.

3.2.2 Multiple Flows with shared tasks

The flows in this case cannot be treated independently since the shared tasks need to be allocated to the same stage (third constraint). However, the operations at each stage are almost the same. At each stage, again there are two initial starting points depending on the previous stage allocations. After making the selections using Equation 3.5, each selection is combined with the selections of the other flows which leads to 2^N combinations for N flows, and the "best" two of these combinations are selected. Thus, the complexity of each stage operation would be $O(2^N)$ and the complexity of execution over all the stages will be $O(2^N R)$.

The space complexity for both the cases is of the same order as their time complexity. A performance comparison in terms of complexity, for all the algorithm approaches discussed in this thesis will be done in Chapter 7.

3.3 *GreedyPipe* Performance

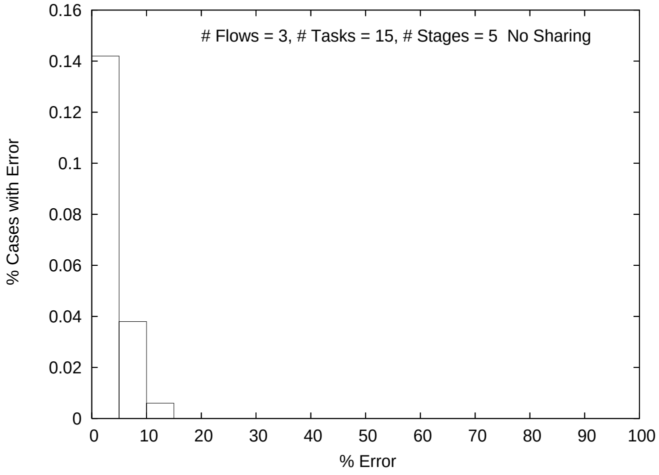


Figure 3.1: % Error vs % Cases with Error

There are two elements associated with evaluating *GreedyPipe* performance. The first concerns how closely *GreedyPipe* results match the true optimal results. While no analytic bounds on the errors have been developed, extensive experimentation has been performed where the results of *GreedyPipe* were compared with the true optimal as obtained by running the time consuming complete enumeration method explained in Chapter 2. For each experiment, the task times were randomly selected from a uniform distribution ranging from 0 to 10 time units. Figure 3.1 shows the results for a system with 3 flows and 10 tasks per flow with no sharing between the tasks. The pipeline consisted of 5 stages and the task to stage assignment was done using *Greedypipe*. A total of 500 experiments were conducted on the described system where, as mentioned before, the task times for each experiment were generated randomly. The figure shows that the error is never more than 15% of the optimal and that less than 1% of results produced by *Greedypipe* were non-optimal.

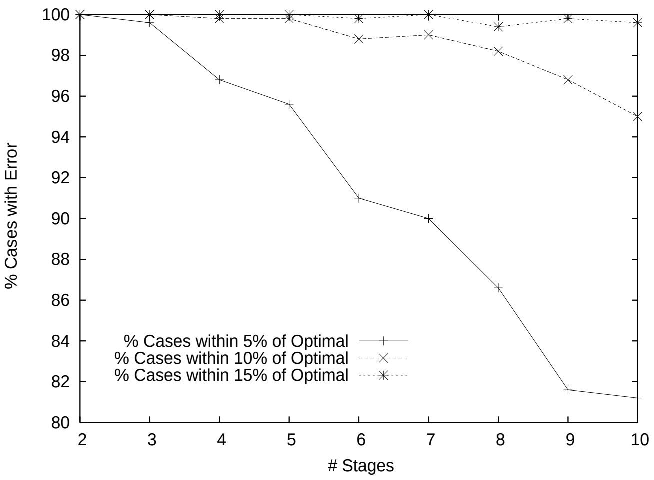


Figure 3.2: % Error vs Number of Stages

Figure 3.2 shows the effect of increase in number of stages on the performance of a system. The system consisted of 3 flows and the number of tasks in each flow was equal to three times the number of stages with no shared tasks between the flows. The number of stages was varied from 2 to 10 and 50 experiments were conducted for each configuration. The results show that the number of non-optimal cases increases with the increase in the number of stages. Almost all experiments result in an error less than or equal to 15% of the optimal. With more stages, the heuristic selection at the earlier stages gets propagated to longer pipeline depths and thus the probability of it leading to an overall optimal solution is reduced.

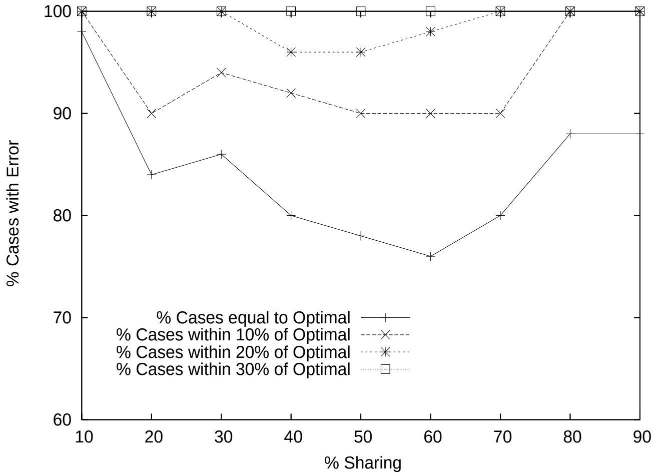


Figure 3.3: % Error vs % Sharing

Figure 3.3 shows another set of experiments on a system with 3 flows and 9 tasks in each flow with tasks shared between the flows. The percentage sharing of tasks between flows was varied from 10% to 90% and the tasks were assigned to a 3 stage pipeline using *Greedypipe*. The results were averaged over 50 experiments for each configuration. The figure shows that up to a point, the percentage of cases with error increase with the increase in sharing of tasks. With tasks shared between flows, the flows cannot be treated independently and at each stage results for each flow are combined with that of other flows to select a probable allocation for that stage. This leads to more number of choices available at each stage and thus the probability of the selection at a stage leading to an overall optimal allocation reduces. Beyond 60% sharing, the percentage of cases with error decreases with sharing since with extensive sharing, there are very few valid allocations possible.

Overall, over a wide range of randomly generated conditions, 98% of the time *GreedyPipe* results are within 15% of the optimal and in no case was the *GreedyPipe* result greater than 25% from the optimal. The results however varied with the number of stages in the pipeline and the percentage of shared tasks associated with different flows.

The experiments were conducted for systems containing 1 to 3 flows, task sharing varying from 0% to 75% and the number of tasks per flow being equal to three times the number of stages. The number of stages were varied from 1 to 5 for systems with shared tasks while problems with up to 10 stages were considered for systems with no task sharing between the flows. A set of 50 experiments were conducted for each possible combination of the above described parameters. The overall results are as follows:

- For small systems involving 2 or 3 stages; 95% of the time the optimal solution was obtained and 98% of the time the result was within 10% of the optimal.
- For larger systems involving 4 or 5 stages; 72% of the time the optimal solution was obtained and 96% of the time the result was within 10% of the optimal.
- For all systems, when the percentage of task sharing was 25% or less, nearly 100% of the time the result was within 10% of the optimal. When there was no task sharing, more than 99% of the time *GreedyPipe* obtained the optimal result.

The second aspect of performance, execution time, is discussed in Chapter 7.

Chapter 4

Dynamic Programming

Dynamic Programming (DP) is an important optimization technique [6]. It is efficient in finding optimal solutions for cases with many overlapping subproblems. It solves problems by successively recombining solutions to subproblems and sub-subproblems. In order to avoid solving these sub-subproblems several times, their results are gradually computed and memorized, starting from the simpler problems, until the overall problem itself is solved. Thus, dynamic programming is simply memorization of results of a recurrence, so that time is not spent trying to solve the same subproblem (or problem) repeatedly. Dynamic programming can only be applied when the problem under concern has optimal substructure [19]. Optimal substructure means that the optimal solutions of local problems can lead to the optimal solution of the global problem. In simple terms, that means that the problem can be solved by breaking it down and solving the simpler problems.

In a multi-processor system, if each processor has a separate data memory module associated with it, there will be no memory contention overheads. However, if the number of memory modules is less than the number of processors, the typical case, memory modules will be shared between processors and the overhead due to memory contention needs to be considered when obtaining the optimal allocation. As described in Chapter 2, memory contention overhead can be calculated only after the complete allocation of tasks is known. In this chapter, we describe the application of DP to the task scheduling problem, however since DP divides the problem in to smaller subproblems and deals with one stage at a time it is not possible to consider memory contention while obtaining the optimal solution. Finding the optimal task **and** memory module assignment is considered in Chapter 5 using Simulated Annealing.

4.1 Dynamic Programming for Task Scheduling

As indicated above, we consider here a simpler form of the scheduling problem such that each processor has a different memory module allocated to it. Thus, there are no contention overheads and, with $c_{jk} = 0$, the performance metric (Equation 2.5) is reduced to:

$$P = \max_{j=1}^{N} P_j = \max_{j=1}^{N} \left\{ \max_{k=1}^{K} \left\{ \sum_{i=1}^{M_j} X_{ijk} t_{ij} \right\} \right\} \tag{4.1}$$

We further simplify the problem to a single flow, F_1, with tasks

$$T = \{T_1, T_2, T_3, ..., T_M\}$$

where M tasks are to be assigned to a pipeline of R processors.

This problem can also be viewed as a partitioning problem where M consecutively ordered tasks are to be partitioned into R intervals such that the maximum time needed to execute the tasks in any interval is minimized. A. Bokhari proposed a dynamic programming solution for this problem in [36] which was further refined by Anily and Federgruen in [33] and Manne and Olstad in [10].

We start by describing the algorithm in [33]. To apply dynamic programming, the optimization problem is first divided into multiple subproblems. Let f be a function that gives the delay associated with the tasks allocated to a stage such that

$$f(i,j) = \sum_{l=i}^{j} t_l \geq 0 \tag{4.2}$$

for $1 \leq i, j \leq M$, with equality if and only if $j < i$. Let $d(j,k)$ represent the maximum processing stage delay over all R stages when tasks $\{T_1, T_2, ..T_j\}$ in a flow are assigned to k stages where $1 \leq j \leq M$ and $1 \leq k \leq R$. Then $d(M,R)$ will give the optimal delay of allocating M tasks to R stages (i.e., a task assignment that minimizes the maximum delay). From Equation 4.2, the cost of assigning the first j tasks to a single stage is given by :

$$d(j,1) = f(1,j) = \sum_{i=1}^{j} t_i \tag{4.3}$$

For any value of j, $1 \leq j \leq M$, $d(j,k)$ can be computed for $2 < k \leq R$ by using the recursion given by Equation 4.4.

$$d(j,k) = \min_{i=k-1}^{j-1} (max[d(i,k-1), f(i+1,j)]) \tag{4.4}$$

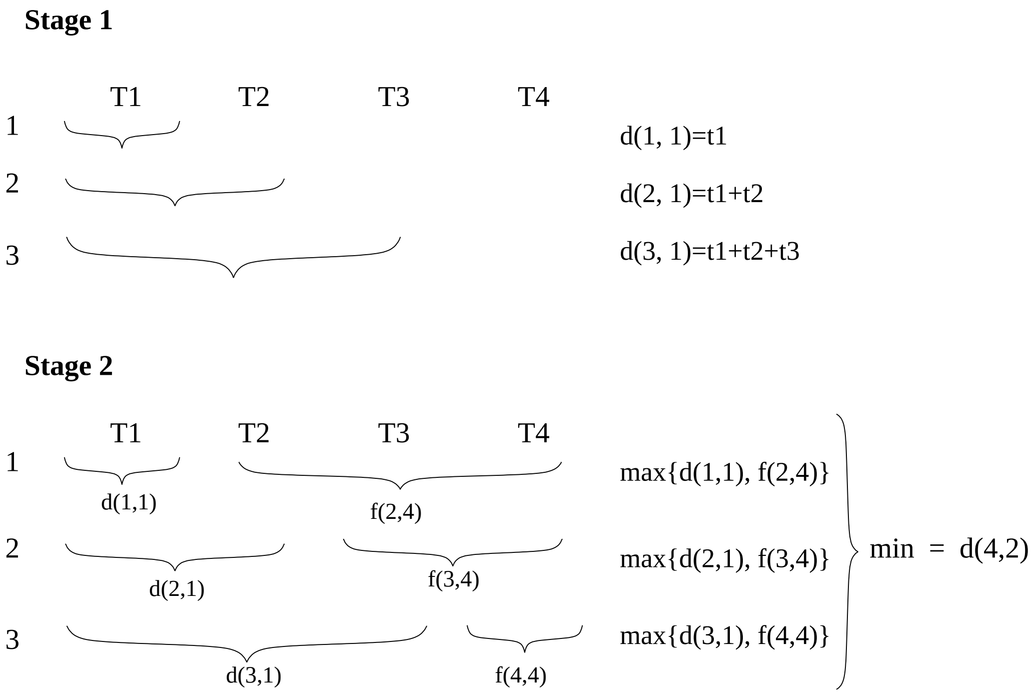


Figure 4.1: Dynamic Programming

Thus, as shown by Equation 4.4, the problem is divided with respect to stages and each stage can have multiple states (for different values of j) where each *state* is defined by the number of tasks allocated up to that stage. Each state at each stage represents an optimal allocation. Note that the optimal solutions at stage k are dependent only on the optimal solutions at stage $k-1$. Thus, given the states at the current stage, the optimal decision for the next stage does not depend on the previous stages but on the states at the current stage. There exists a recursive relationship that identifies the optimal decision for stage k, given that stage $k-1$ has already been solved, so that $d(M,R)$ can be computed by finding the values for all the states for stages $1 \leq k \leq R$. Since $d(M,R)$ gives only the optimal delay, the actual task allocations associated with it can be obtained by reverse traversal of the states in the previous stages. The lower limit on i is set to $k-1$ to ensure that there is atleast one task allocated to each stage.

Equation 4.4 can be better explained using the example in Figure 4.1. The example illustrates application of DP to the tasks in a single flow to be allocated to two stages. The flow consists of four tasks, T_1, T_2, T_3 and T_4, with their execution times given by t_1, t_2, t_3 and t_4 respectively. At the first stage, there are four possible states depending on the number of tasks selected (i.e., $d(1,1)$, $d(2,1)$, $d(3,1)$ and $d(4,1)$). The figure does not show the fourth state, $d(4,1)$, since if all the four tasks are allocated to the first stage, stage 2 will be empty.

Consider next the second stage. At this stage, all the algorithm requires is the knowledge of the number of tasks already allocated in the previous stages (in this case just the first stage) and the optimal value for the allocations. Accordingly, the optimal state values at stage 1 are used to obtain the values for all the possible states at stage 2. At stage 2, there are 4 possible states, $d(1,2)$, $d(2,2)$, $d(3,2)$, $d(4,2)$. However, the algorithm selects only the value of $d(4,2)$ since it is the optimal delay when all the 4 tasks are allocated to 2 stages. The figure shows the possible allocations at stage 2 for the state $d(4,2)$. As the figure indicates, $d(4,2)$ is obtained by selecting the allocation that minimizes the maximum delay which is equivalent to obtaining the minimum value by using the Equation 4.4 as :

$$d(4,2) = \min_{i=2-1}^{4-1} (max[d(i,2-1), f(i+1,4)]) \tag{4.5}$$

Thus, if $d(4,2) = \max\{d(3,1),\ f(4,4)\}$, then allocating the first three tasks to the first stage and the fourth task to the last stage gives an optimal solution.

For multiple flows, the same approach can be applied as long as there are no shared tasks between the flows. Since there is no dependency between the flows, tasks in each flow are assigned independently of those in the other flows. Thus, if there are N flows, the problem is reduced to solving N problems independently, and combining the results to get the optimal allocation for N flows. The next section demonstrates application of dynamic programming to the task scheduling problem using an example.

Table 4.1: A Single Flow with five *ordered* tasks

	Task 1	Task 2	Task 3	Task 4	Task 5
Flow 1	T_1	T_2	T_3	T_4	T_5
Task Execution Times	2	4	3	1	3

4.1.1 Example

A simple single flow example with five tasks and a three stage pipeline is now presented. The task times are as given in Table 4.1. The algorithm starts by calculating $d(j,1)$ for $1 \leq j \leq 5$. Thus using Equation 4.3, for $k = 1$, the optimal solutions at *Stage 1* are as shown in Table 4.2.

Table 4.2: Optimal Solutions at Stage 1

j	f(1,j)	d(j,1)
1	2	d(1,1)=2
2	2+4=6	d(2,1)=6
3	2+4+3=9	d(3,1)=9
4	2+4+3+1=10	d(4,1)=10
5	2+4+3+1+3=13	d(5,1)=13

Subsequently, Equation 4.4 is use to obtain the optimal solutions at *Stage 2* from those at *Stage 1*. Thus, at *Stage 2*, there will be three possible states corresponding to $2 \leq j \leq 4$ and $k = 2$, given by

Table 4.3: Possible States at Stage 2

j	$d\ (\ j,\ 2)$
2	$\min{}_{i\ =1}^{2-1}\ (\ max\ [\ d(i,\ 1),\ f(i+1,\ 2)\ ]\ )\ )$
3	$\min{}_{i\ =1}^{3-1}\ (\ max\ [\ d(i,\ 1),\ f(i+1,\ 3)\ ]\ )\ )$
4	$\min{}_{i\ =1}^{4-1}\ (\ max\ [\ d(i,\ 1),\ f(i+1,\ 4)\ ]\ )\ )$

The optimal values associated with each state at *Stage 2* are as shown in Table 4.4. $d(5,2)$ need not be computed at *Stage 2* since at least one task needs to be left unassigned for *Stage 3*. If T_5 is allocated to *Stage 2*, *Stage 3* will have no tasks assigned to it.

Table 4.4: Optimal State Values at Stage 2

j	i = 1	i = 2	i = 3	d (j , 2)
2	max [d(1,1), f(2,2)] =max [2,4] = 4			d(2,2) = 4
3	max [d(1,1), f(2,3)] =max [2,7] = 7	max [d(2,1), f(3,3)] =max [6,3] = 6		d(3,2) = 6
4	max [d(1,1), f(2,4)] =max [8,2] = 8	max [d(2,1), f(3,4)] =max [4,6] = 6	max [d(3,1), f(4,4)] =max [1,9] = 9	d(4,2) = 6

Similarly the states and the optimal values associated with the states for *Stage 3* are shown in Table 4.5 and 4.6.

Table 4.5: Possible States at Stage 3

j	$d\ (\ j,\ 3)$
3	$\min{}_{i\ =2}^{3-1}\ (\ max\ [\ d(i,\ 2),\ f(i+1,\ 3)\ ]\ )\ )$
4	$\min{}_{i\ =2}^{4-1}\ (\ max\ [\ d(i,\ 2),\ f(i+1,\ 4)\ ]\ )\ )$
5	$\min{}_{i\ =2}^{5-1}\ (\ max\ [\ d(i,\ 2),\ f(i+1,\ 5)\ ]\ )\ )$

We focus only on the value of $d(5,3)$ since that gives the cost of optimally allocating all the 5 tasks to 3 stages.

Now that the optimal delay is obtained, the stages need to be traversed in a reverse order to obtain the task allocation associated with the optimal delay. At stage

Table 4.6: Optimal State Values at Stage 3

j	i = 2	i = 3	i = 4	d(j,3)
3	max [d(2,2), f(3,3)] =max [4,3] = 4			d(3,3) = 4
4	max [d(2,2), f(3,4)] =max [4,4] = 4	max [d(3,2), f(4,4)] =max [6,1] = 6		d(4,3) = 4
5	max [d(2,2), f(3,5)] =max [4,7] = 7	max [d(3,2), f(4,5)] =max [6,4] = 6	max [d(4,2), f(5,5)] =max [6,3] = 6	d(5,3) = 6

3, there are two states that lead to the optimal delay for values of $i = 3$ and $i = 4$. The optimal allocations at stage 3 are associated with the values $d(3, 2)$ and $d(4, 2)$ at stage 2. Thus, there are two optimal allocations obtained from Tables 4.6 and 4.4, each with the same maximum delay of $T = 6$. The optimal assignments are shown in Table 4.7.

Table 4.7: Optimal allocations

Allocations	Stage 1	Stage 2	Stage 3
1	$T_1, T_2, \quad T = 6$	$T_3, \quad T = 3$	$T_4, T_5, \quad T = 4$
2	$T_1, T_2, \quad T = 6$	$T_3, T_4, \quad T = 4$	$T_5, \quad T = 3$

The next section discusses a variation in the dynamic programming algorithm to address the problem of dealing with shared tasks between multiple flows.

4.1.2 Multiple Flows with Shared Tasks

The problem becomes much more complex when there are tasks shared between the flows, since the flows cannot be treated independently. Let us denote $d(j, k)$ for flow F_1 as $d_{F_1}(j, k)$, for flow F_2 as $d_{F_2}(j, k)$ and so on. Similarly, let $f(j, k)$ for flow F_1 be denoted by $f_{F_1}(j, k)$, for flow F_2 be denoted by $f_{F_2}(j, k)$ and so on. Equation 4.4 in Section 2.1 is now modified for multiple flows with shared tasks. For two flows the equation is given by :

$$d(j_1, j_2, k) = \min_{i_1=1}^{j_1}[\min_{i_2=1}^{j_2}(max[d_{F_1}(i_1, k_1 - 1), f_{F_1}(i_1 + 1, j_1), d_{F_2}(i_2, k_2 - 1), f_{F_2}(i_2 + 1, j_2)])] \tag{4.6}$$

Equation 4.6 indicates that at each stage, there will be $j_1 \times j_2$ states and every state is computed by combining each possible solution for flow F_1 with that of flow F_2. Also the limits in this case change since we may not want to force allocating at least one task of each flow to each stage. When the *max* value in the equation is evaluated, the validity of the allocation is checked, so that a shared task in different flows should not be allocated to different stages. Consider a case with 2 flows, F_1 and F_2 with shared tasks. f_{F_1} now includes a shared task at some stage k while f_{F_2} does not include the shared task at that stage. This implies that if this allocation is considered valid, the shared task will not be allocated to the same stage. To comply with the allocation constraints, such allocations where the shared task is assigned to different stages are marked invalid and not considered in obtaining the optimal value for a state. The above equation can be extended to include N flows, however for larger values of N the complexity of the solution will be very high.

4.2 Complexity

For convenience, we reproduce Equation 4.4.

$$d(j, k) = \min_{i=k-1}^{j-1}(max[d(i, k - 1), f(i + 1, j)]) \tag{4.7}$$

In Equation 4.7, computation of each value of f can be assumed to have a complexity of $O(1)$. To obtain each value of $d(j, k)$ for $k - 1 \leq i \leq j - 1$, $(j - 1) - (k + 1)$ values of f need to be computed which will lead to a complexity of roughly $O(M)$ for M tasks in a flow. Also, for M tasks in a flow, at each stage $M - k$ states (k less since we do not want to leave any stage empty) will be possible and thus $M - k$ values of d need to be computed at each stage for use at the next stage. Thus, at each stage the complexity for computing all the possible states is given by $O(M^2)$ while the time complexity for a single flow allocation to R stages is $O(M^2R)$. Accordingly, the time complexity for N *independent* flows to be allocated to R stages will be given by $O(M^2RN)$.

For two flows with shared tasks, from Equation 4.6, each value of d requires computation of $j_1 \times j_2$ values of the function f. Thus the complexity of computing each value of d is given by $O(M^2)$ for M tasks in each of the two flows. Again, there will be $j_1 \times j_2$ states possible at each stage. Therefore, the time complexity for R stages for the two flows is given by $O((M^2)^2R)$. Consequently, for N flows, there will be $j_1 \times j_2 \times j_3 ... j_N$ values of d computed at each stage and the same number of states also will be possible at each stage. Thus, the time complexity for N flows will be given by $O((M)^{2N}R)$. A comparison of the complexity of the dynamic programming solution with the other approaches described in the thesis will be presented in Chapter 7.

Chapter 5 describes a statistical approach to solve the task scheduling problem that takes into consideration the overheads due to memory sharing.

Chapter 5

Simulated Annealing - A Statistical Approach

Simulated annealing exploits an analogy between the way in which a metal cools and finally freezes [29] (the process of annealing) achieving a minimum energy crystalline structure, and the search for a state minimizing a specified performance metric or objective function in a general system. In an annealing process, an initial state of a system, approximately in thermodynamic equilibrium is chosen at energy E and temperature T. Holding T constant, the initial configuration is perturbed and the change in energy dE is computed. If the change in energy is negative the new system state after perturbation is accepted. If the change in energy is positive, the new system state is accepted with a probability given by the Boltzmann factor $e^{-(dE/T)}$. For the current temperature, this process is then repeated a sufficient number of times to come close to the minimum energy state for that temperature. The temperature is then decremented and the entire process repeated until the system approaches the lowest energy level possible at T=0 where, in a real annealing environment, the material becomes a single crystal.

Simulated Annealing, a generalization of this process to combinatorial optimization problems, was first proposed by Kirkpatrick in [34]. It attempts to minimize

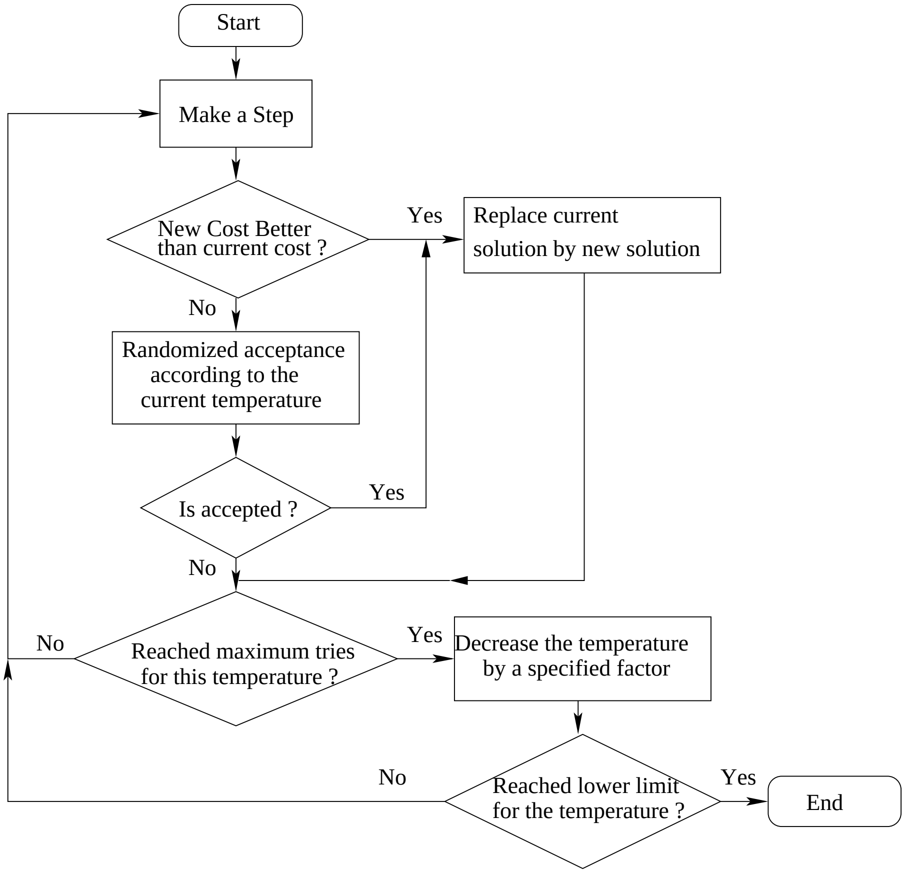

Start
Make a Step
New Cost Better than current cost ?
Yes
Replace current solution by new solution
No
Randomized acceptance according to the current temperature
Is accepted ?
Yes
No
No
Reached maximum tries for this temperature ?
Yes
Decrease the temperature by a specified factor
No
Reached lower limit for the temperature ?
Yes
End

Figure 5.1: Simulated Annealing Flow Diagram

a given objective function over the entire system state space. Solutions in a combinatorial optimization problem are analogous to states in a thermodynamic system, the cost of a solution is equivalent to the energy of a state and ground state (i.e., state at which T=0) is analogous to the global minimum of the objective function. The major difficulty in implementation of the algorithm is that there is no obvious analogy for the temperature T with respect to a free parameter in the combinatorial problem. Furthermore, it's ability to avoid local minima is dependent on the "annealing schedule"; the choice of initial temperature, how many iterations are performed at each temperature, and how much the temperature is decremented at each step as cooling proceeds. Figure 5.1 shows the flow diagram for a generic simulated annealing process.

5.1 Simulated Annealing for Task Scheduling

For the task/pipeline system in question one must first define a set of states. In this problem domain that set constitutes all the possible allocations of tasks to pipeline stages that satisfy system constraints (e.g., maintaining task ordering). To implement the algorithm we must now decide on the following items

- Step Change : a generator of random changes in task-to-stage allocations.
- Annealing Schedule : rules for lowering the temperature as the search progresses.
- Initialization : initial allocation of tasks and inital temperature.

5.1.1 Step Change

A *step* constitutes a move from one state to another. A *step* must be defined so that over a set of random steps, if the number of steps is sufficiently large, it must be possible to reach every state in the system.

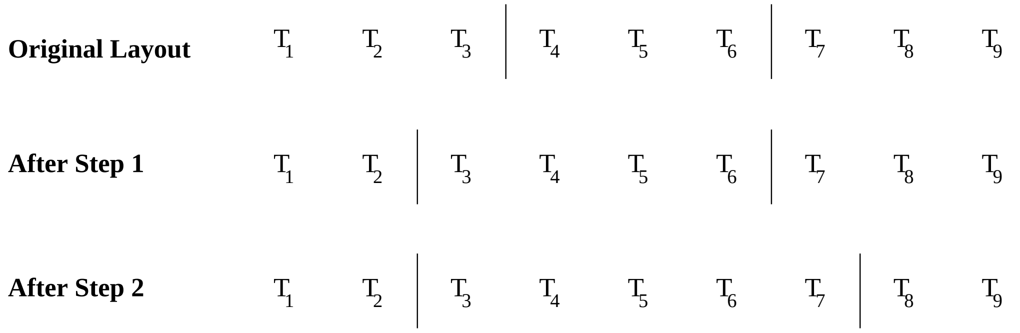

Original Layout
T_1 T_2 T_3 | T_4 T_5 T_6 | T_7 T_8 T_9
After Step 1
T_1 T_2 | T_3 T_4 T_5 T_6 | T_7 T_8 T_9
After Step 2
T_1 T_2 | T_3 T_4 T_5 T_6 T_7 | T_8 T_9

Figure 5.2: A Step in Simulated Annealing

In the system considered here we assume that we are given a memory to stage assignment and restrict ourselves to steps that involve the movement of tasks from one stage to another. Another approach is to include the memory assignment as a part of the state definition. This was not considered in this thesis. Thus, for example, Figure 5.2 shows a system with a single flow consisting of nine tasks and a three stage pipeline. Two successive steps are illustrated. The two vertical lines delineate the tasks associated with each stage. The first step moves T_3 from stage 1 to stage 2 while the second step moves T_7 from stage 3 to stage 2. Note that a step in this system involves randomly moving one stage boundary task from one stage to an adjacent stage. By restricting the step in this manner, task ordering is maintained. After each step, the throughput associated with the given task assignment and memory configuration is evaluated. If the throughput is greater, then the step has moved the system closer to an optimum value. Another step is then taken and the process repeated.

When a step results in a decrease in throughput, then another aspect of the simulated annealing algorithm is employed. In order to guarantee that the entire state space is explored and that a global optimum is found, under certain circumstances, a step that results in a lower throughput is accepted. This acceptance of a less optimum state is determined if the inequality shown below is satisfied.

$$p < e^{(NewThroughput - PreviousThroughput)/Temp} \tag{5.1}$$

p is a random number between 0 and 1 that is generated as part of the algorithm. $Temp$ (temperature) is an algorithm parameter that decreases over the course of the optimization procedure. Thus, for each step that results in a lower throughput, p is generated and if the inequality is satisfied, the step and associated new state is accepted. The process then repeats itself through N_{step} steps (discussed in Section 5.1.2) before the temperature is decreased. A sequence of N_{step} steps is referred to as a temperature cycle.

5.1.2 Annealing Schedule

As the optimization procedure proceeds, $Temp$ is periodically lowered so that the probability of accepting lower throughput states diminishes (i.e., $Temp_{i+1} < Temp_i$). To ensure that each state is reachable and that one does not get trapped in the local minima, the initial temperature $Temp_1$ should be set at a reasonably high value. We have chosen this value to be fifty times of the difference in throughputs over a *Step Change*. Thus, after the initial task assignment (described in Section 5.1.3), the throughput, $Throughput_{init}$, is calculated. Subsequently, a *Step Change* is made which leads to a new throughput, $Throughput_{new}$, for the system. Then, the value of initial temperature is set to :

$$Temp_1 = |Throughput_{init} - Throughput_{new}| \times 50$$

The number of temperature cycles is $T_{cycles} = 10R^2NM$ where R is the number of stages, N is the number of flows and M is the maximum number of tasks across all flows. After each temperature cycle is completed, $Temp$ is reduced to 90% of its prior value (e.g., $Temp_i = 0.9 \times Temp_{i-1}$). After proceeding through all the temperature cycles, the annealing/optimization process is complete. As the simulated annealing algorithm goes through temperature cycles, the probability of accepting less optimal allocations keeps decreasing. While the deviation in throughputs of accepted solutions is high during the initial temperature cycles, it decreases as the temperature reduces and the solution converges towards the optimal. Thus, for lower temperatures, when the difference between throughputs for a set of consecutive temperature cycles is very low, the solution is likely to be an optimal solution. Given the above, an additional terminating condition is introduced so that the simulated annealing ends if the following condition is satisfied.

$$\begin{aligned}
&(T_{cycles} > 1000)\ AND \\
&(Throughput_i - Throughput_{i-1}) < (0.001 \times Throughput_i)\ AND \\
&(Throughput_{i-2} - Throughput_{i-3}) < (0.005 \times Throughput_{i-2})\ AND \\
&(Throughput_{i-4} - Throughput_{i-5}) < (0.01 \times Throughput_{i-4})
\end{aligned}$$

$Throughput_i$ refers to the throughput at the end of the i^{th} temperature cycle. Often, the optimum is obtained with fewer than $T_{cycles} = 10R^2NM$. The additional termination condition helps in reducing overall execution time while still obtaining near optimal results.

The number of steps, N_{step}, per temperature cycle is also reduced from cycle to cycle. Initially the first temperature $N_{step,1} = 10R^2NM$. $N_{step,i}$ is reduced at the end of each temperature cycle following a logarithmic formula (Equation 5.2) so that $N_{step,T_{cycles}}$ is equal to $N_{step,1}/10$.

$$N_{step,i+1} = N_{step,i} * e^{(\log((N_{step,1}/10)/N_{step,1})/(T_{cycles}-1))} \tag{5.2}$$

Thus, as the temperature decreases both the probability of acceptance of states that have lower throughput is decreased, and the number of states examined at a given temperature is decreased. The procedures described above are referred to as the annealing schedule and the parameters associated with the schedule have been arrived at by evaluating the accuracy of the results over a range of experiments.

5.1.3 Initial Task Assignment

Before applying the simulated annealing process, an initial allocation for tasks should be made. The tasks should be reasonably scattered across stages for faster convergence to an optimal allocation. The tasks are assigned to stages using the dynamic programming algorithm described in Chapter 4. In case of multiple flows with shared tasks, after applying the dynamic programming algorithm, the task allocations are validated to make sure that the allocation satisfies the allocation constraints (a shared task is allocated to the same stage). If the constraints are not satisfied, a simple algorithm is used which forces the shared task in different flows to be allocated to the same stage.

5.1.4 Multiple Flows

For multiple flows with no shared tasks, a *Step* is applied to each of the flows independently but the maximum delay within a *Step* is calculated across the flows. The flows cannot be dealt with independently since the worst case memory contention is dependent on the allocation of all the flows. In the case of multiple flows with shared tasks, in addition to the above, within a *Step*, the validity of the assignment is checked to verify if it satisfies the sharing constraints.

5.2 Complexity

The complexity of the simulated annealing algorithm is primarily determined by the annealing schedule. We first need to compute the complexity for each step change of the algorithm. Other than the random perturbation, which can be assumed to have a complexity of $O(1)$, a step involves computation of the memory contention delay and the maximum stage delay. The maximum contention delay for tasks of flow j allocated to stage k is obtained using Equation 5.3 (discussed in Appendix A).

$$c_{jk} = \sum_{p=1, p \neq k}^{R} z_p \qquad \text{where } z_p = \begin{cases} mem_p & \text{if } mem_p < mem_{jk} \\ mem_{jk} & \text{if } mem_p \geq mem_{jk} \end{cases} \tag{5.3}$$

From the equation, for a pipeline with R stages and a single memory module, it requires R iterations to calculate the contention at each stage and the contention needs to be calculated for all the stages in the pipeline. The maximum stage delay is also computed along with the memory contention in the same iteration. Thus, the complexity of computations at each step is $O(R^2)$. We go through approximately $10R^2NM$ steps during each temperature cycle and a maximum of $10R^2NM$ number of temperature cycles. Thus, the total complexity of obtaining a solution to the task allocation problem using simulated annealing is given by $O((R^2)(R^2NM)^2)$. However, the additional terminating condition significantly reduces the number of iterations that the algorithm needs to go through. For smaller problems with R=4, N=3 and M=8, the simulated annealing is terminated almost after one third of the total number of iterations have completed (reduction by a factor of 10^6).

5.3 Performance and Constraints

The first aspect of performance concerns the accuracy of the algorithm. The annealing schedule of the simulated annealing algorithm plays a significant role in determining how close the results are to the optimal solution. As mentioned earlier, the parameters associated with the annealing schedule have been arrived at empirically. The results have been tested extensively and compared with results obtained using exhaustive search and, for more limited cases, with dynamic programming techniques. They indicate that with this schedule we obtain optimum assignments more than 99.5% of the time with the worst case assignment being within 30% of optimal. The other element of concern is the execution time of the algorithm which is again primarily determined by the annealing schedule. The performance can be improved by varying the parameters associated with the annealing schedule, however, it may affect the accuracy of the algorithm.

Chapter 6

Pipeline Design

This chapter illustrates how the task scheduling algorithms can be used as design aids in situations where the effects of pipeline depth, task sharing, task partitioning, number of pipelines and memory to processor assignment are to be explored as part of the design process. Initially, they are used for a generic case employing synthetic data and subsequently they are applied to a specific NP problem where the applications include routing, compression and encryption.

6.1 Pipeline Design using Scheduling Algorithms

In systems, such as Network Processors, with multiple pipelines and flows, determining the best pipeline and algorithm partitioning and pipeline stage assignment or memory stage assignment is difficult. The designer typically has a number of trade-offs to consider. These include:

- **Number of Pipeline Stages and Number of Pipelines:** Given applications, and associated flows, that have been partitioned into a number of ordered tasks, a designer can select the number of pipeline stages to implement. Up to a point, more stages generally result in higher throughput, however, more stages also requires more chip area and high power consumption. Te described scheduling algorithms can be used to determine just what throughput can be achieved with a varying number of stages and pipelines.

- **Algorithm Task Sharing:** When multiple flows and associated applications are present, there is often an opportunity for the sharing of applications or of individual tasks across flows. This may result in smaller overall code space being required which, in turn, may reduce the cost of on-chip memory, or increase performance due to reduced instruction cache miss rates. However, when tasks are shared, there is less flexibility in task-to-stage assignments and generally lower overall throughput results. Use of task scheduling algorithms permits fast determination of the performance effects related to task sharing.

- **Algorithm Partitioning:** For many applications, alternative algorithm to task partitionings are possible. For a given pipeline, each partitioning, after assignment, generally leads to different throughput results. The scheduling algorithms can be used to determine those tasks that are performance bottlenecks and what performance gains can accrue from task repartitioning. Up to a point, for a fixed pipeline design, this may result in higher throughput, however, at the cost of algorithm and software redesign.

- **Number of Memory Banks and Memory Bank Assignment:** Fewer memory banks lead to increased memory contention and may affect the performance significantly for memory intensive applications. Increasing the number of memory banks in the system improves the performance to a certain point. However more memory banks requires more chip area (if the memory is on-chip) and may result in increased cost for the system too. The scheduling algorithms can be used to determine the optimal number (from the perspective of throughput) of memory banks and their effective assignment to processors.

The sections that follow illustrate the use of scheduling algorithms in these sorts of design activities. In each subsection, figures illustrating the results of a number of experiments are provided. Each data point presented represents the results of averaging forty experiments. In each experiment the task times were randomly selected from a uniform distribution ranging from 0 to 10 time units.

6.1.1 Number of Pipeline Stages and Number of Pipelines:

The DP approach was used to determine the effect of pipeline depth on throughput performance. This is illustrated in Figure 6.1 where the results for a system with 6 flows, and 20 tasks per flow is shown.

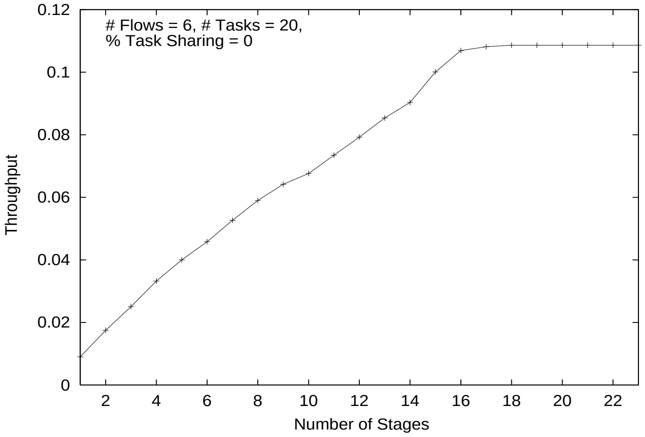


Figure 6.1: Throughput vs number of pipeline stages

As expected, the throughput increases with the number of stages, and with this many flows and tasks, the increase is almost linear until one reaches about sixteen stages. After that, it is more difficult to evenly distribute the tasks over the stages and the throughput asymptotically approaches a maximum near 0.1 (i.e., $1/(maximum\ task\ time)$). This maximum is a result of the fact that it is likely that there is at least one task generated with a value near the maximum time of 10 (i.e., the floating point random number generator could yield a value of, for example, 9.7 which would lead to a maximum of 0.103). Figure 6.2 shows the results obtained when the throughput was plotted as a function of the number of pipelines and the stages in the pipeline. The experiment consisted of 6 flows with 10 tasks in each flow and the plots were generated by varying the number of pipelines and the number of stages in the pipelines. With multiple pipelines, the system throughput is equal to the sum of the throughputs of all the pipelines. To find the optimal allocation of flows to pipelines, all possible ways of assigning the flows to the pipelines were evaluated and the allocation that maximized the system throughput was selected as the optimal allocation. As expected, the throughput increases with the increase in number of pipelines and stages. When the number of pipelines equals 6, one flow is assigned to each pipeline.

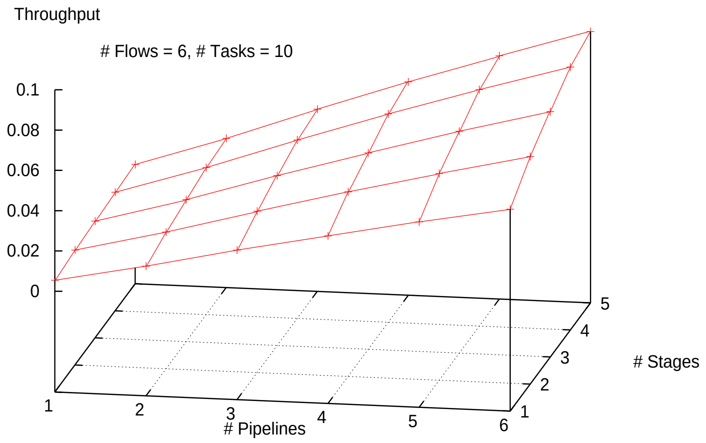


Figure 6.2: Throughput vs number of pipelines vs number of stages

6.1.2 Sharing of Tasks Between Flows

Task sharing between flows leads to an interdependence between the flows. This may be advantageous and result in better memory utilization and lower instruction cache miss rates, however, it also restricts the number of assignment options and thus potentially reduces the maximum throughput.

Experiments were conducted for the case of 6 flows, 20 tasks per flow and a single 8 stage pipeline where the fraction of tasks for each flow that are shared with other flows was varied. Thus, a 50% level of sharing means that 50% of tasks for each of the flows are common with the other flows. Due to high time and space complexity of the DP approach for larger problems with shared tasks, *GreedyPipe* was used to obtain the task allocations. As expected, the results (Figure 6.3) indicate a significant decrease in throughput as more tasks are shared between flows. The decrease is over 35% when one moves from 0% sharing to 100% sharing. In a full design analysis, this would be balanced against the potential gains noted above.

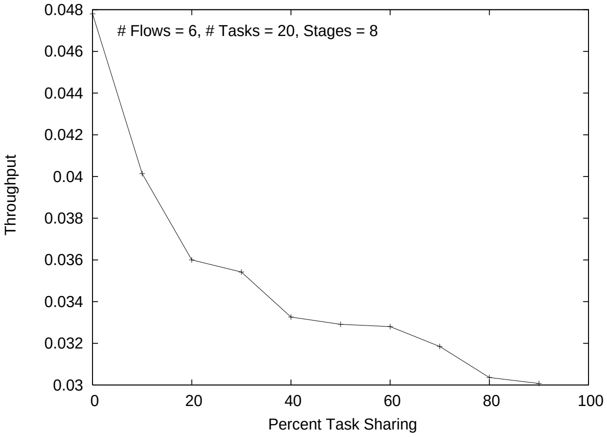


Figure 6.3: Throughput vs percent shared tasks

6.1.3 Task Partitioning

For many applications that can be implemented in a pipelined manner, there is a choice concerning the task partitioning of the application. Having more tasks generally results in both having greater flexibility in assigning the tasks to the hardware pipeline and in being able to use longer pipelines. This usually results in higher throughput. However, there are two potential drawbacks. First, it can be difficult to divide tasks beyond some basic application partitioning and thus there may be a nontrivial personnel cost associated with this job. Second, greater task partitioning often results in larger inter-task communications costs that may increase latency and reduce throughput. However, in order to make a judgement as to whether increased task partitioning is worthwhile considering, it is first necessary to determine the potential performance gains that might result from such an endeavor. DP was used to examine the possible gains from additional task partitioning.

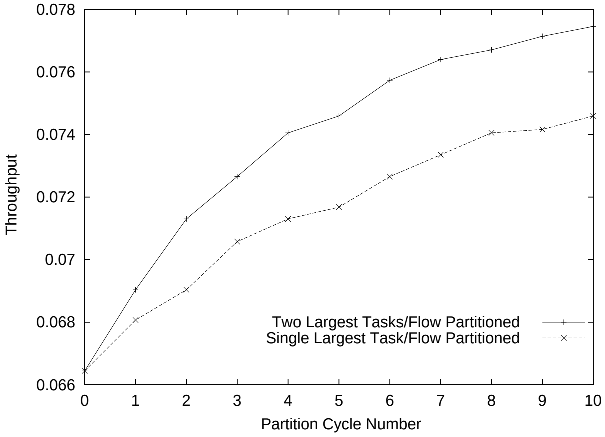


Figure 6.4: Throughput vs task partitioning

The effects of task partitioning on throughput are illustrated in Figure 6.4. For both curves presented, the experiments had 2 flows, 11 tasks per flow, a single 5 stage pipeline, and no task sharing. With the lower curve, the longest task in each flow is successively divided into two equal tasks and then the DP is used to find a new task assignment. The "Partition Cycle Number" corresponds to how many times this division has occurred (a new maximum task is determined and divided on each cycle). With the upper curve, the two longest tasks in each flow are divided in a similar manner and the throughput is obtained. Both cases are beneficial since both provide more opportunities for improved task assignments that aim at equalizing the delay associated with each stage (and thus maximize throughput). This permits the designer to determine the potential benefit associated with spending more time on algorithm partitioning.

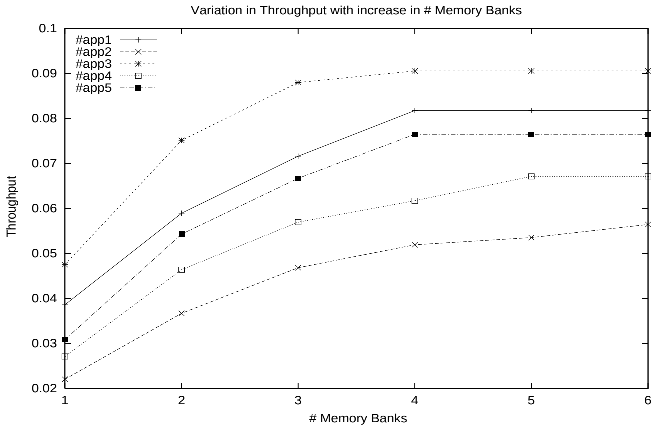


Figure 6.5: Throughput versus Number of Memory Banks

6.1.4 Number of Memory Banks and Memory Bank Assignment

Sharing of memory banks leads to memory contention which effectively reduces the throughput of the system. It may not be always feasible to have a single memory bank per processor since an increase in number of banks increases chip area which in turn results in increased power consumption. In such a case, the memory banks also need to be allocated to the processors optimally. Additionally, increasing the number of memory banks also results in increased performance. A trade-off needs to be attained between the number of memory banks and the cost involved in it. Figure 6.5 shows the results of an experiment using Simulated Annealing, where 5 different randomly generated applications were considered. Each application had a single flow consisting of 12 tasks and 6 stages. The number of memory modules were varied from 1 to 6. Each point represents an optimal allocation of memory banks and tasks. All possible memory to processor allocations were enumerated and the optimal task allocation was obtained for each of them. The memory to processor allocation that minimized the optimal task allocation delay was selected as the optimal allocation for tasks and memory. As expected, the throughput increases with increase in the number of memory banks and when the number of memory banks equals the number of stages, each application attains the maximum throughput. At this point each stage has an independent memory bank allocated for it and thus there is no memory contention.

The next section describes the use of the task scheduling algorithms for a specific problem in the networking domain.

6.2 A Network Processor Problem

Most of the material in this section has been published in [25]. The Intel IXP 2800 is an example of an NP that can potentially be configured as a set of processor pipelines where the total number of processor stages is sixteen. Consider now a workload where there are the following three flows:

1: Longest Prefix Match (LPM) - A flow where the only function to be performed is that of routing the packet using the LPM algorithm.

2: LPM & Compression - A flow where the NP must perform LPM and also compression on the packet payload.

3: LPM & Encryption - A flow where the NP must perform LPM and also encryption on the packet payload.

Software implementations for LPM, Compression and Encryption are available and may provide adequate performance at the edges of the network. However, as one moves towards the core of the network, real-time constraints generally requires the use of fast special purpose hardware. An alternative to providing such special hardware is to utilize a general processor pipeline for these applications. As discussed later, pipelined implementations for each of these three functions are available [40, 28, 23, 30]

Say that our objective is to maximize overall throughput given the number of available pipeline processor stages[1]. Given general pipelined implementations of the

[1] Two important issues are omitted in this discussion. First, we are not considering multi-threading effects. However, if we assume a high enough level of multi-threading so that all off-chip

functions, and their associated task times, the design space for maximizing throughput includes the following choices:

1. Number and length of pipelines: Both the number of pipelines and the length of each pipeline can be selected subject to the constraint that the total number of processors over all the pipelines must be ≤ the number of processors available.

2. Number of tasks per function: Given a general approach to implementing each of the functions as a software pipeline of ordered tasks, just how should a particular set of tasks be selected.

3. Assignment of function tasks to pipeline stages: Given the two items above, assign each of the tasks to the pipeline stages in a manner that maximizes the overall throughput of the NP.

With the use of scheduling algorithms, it is a relatively simple matter to explore key aspects of this design space. Given a pipelined function implementation (item 2) and a choice of number and length of pipelines (item 1), the scheduling algorithms will choose a near optimal assignment of tasks to stages (item 3). One can then iterate over set of allowable pipeline configurations (item 1) and obtain a near optimal overall design. We next review the flow functions and some pipelined implementation options. All the results in the following sections were obtained using *Greedypipe.*

6.2.1 Longest Prefix Matching (LPM)

Most of the material in this subsection has been published in [23]. Performing IP address lookup for packet routing is a key operation that must be performed by routers and such lookups require that a Longest Prefix Matching (LPM) algorithm be executed [28]. Because of its central role[40], NPs often include facilities to perform fast IP prefix matching often using a combination of software and special purpose

memory latencies are masked, then our rough throughput analysis is not significantly effected by this assumption. Second, we are not considering other issues associated with the memory hierarchy. That is, we are assuming that contention for common resources is not appreciable. Extensions to this work will bring these effects into the model.

hardware. One may also implement LPM utilizing a processor pipeline. Such an approach potentially has high and additionally may also be modified to meet the requirements of evolving standards. This section considers a pipelined LPM algorithm based on the work of Moestedt and Sjodin [28]. In their paper, a dedicated pipeline of special purpose hardware is presented. Our implementation uses a pipeline of general purpose processors and is based on the development of a routing tree that contains three types of nodes:

- **Valid Route Nodes**: Tree leaf nodes that correspond to legal or valid routes (or destinations). Associated with these nodes are the router output port information.
- **Invalid Route Nodes**: Tree leaf nodes that correspond to invalid routes.
- **Part Route Nodes**: Tree interior that represent branching nodes in the tree and correspond to part of a prefix.

Consider an example (Figure 6.6) containing three prefixes embedded within a three level tree. The first level, leaf nodes labeled (1), is of length 3 and corresponds to the prefix 001. The second, leaf nodes labeled (2), is of length 7 and corresponds to prefix 0010001. The third, leaf nodes labeled (3) is of length 3 and corresponds to prefix 110. Note that with this LPM algorithm smaller prefixes (e.g., 001) that are themselves contained in longer prefixes (e.g., 0010001) may spawn additional levels and Valid Route Nodes (e.g., level 3 nodes corresponding to the first prefix). Constructing the tree itself requires that one first decide on the number of levels desirable, and the number of bits to be considered at each level. Thus, there are numerous tree structure variants that satisfy the routing table requirements, and the associated data structures result in differing memory requirements.

Given a packet destination address and the above routing tree structure, obtaining the route involves accessing the tree successively and following the path associated with this address. Thus, say we have the address [0010 0111 X] where X

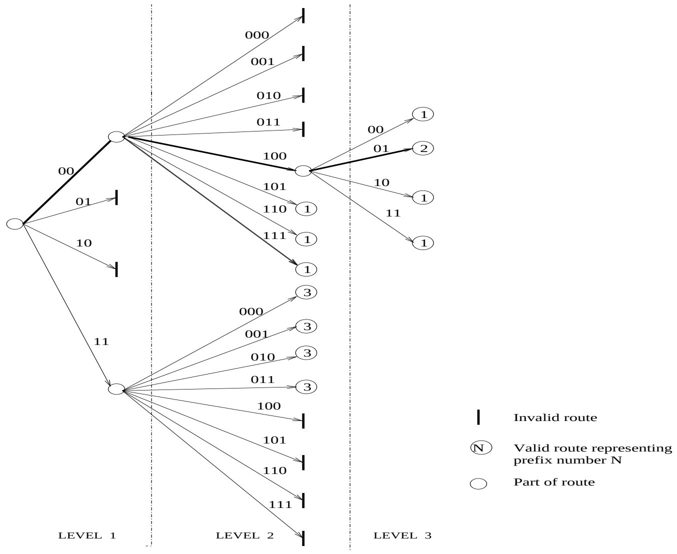


Figure 6.6: Example Longest Prefix Match Tree

corresponds to the remaining 24 bits in a 32-bit address. Reaching leaf node (2) in Level 3 requires three tree lookups following the wide path in Figure 6.6.

An approach to pipelining such an algorithm is to associate each level lookup with a separate task and allocate these tasks to stages in a processor pipeline. In the above example a three processor pipeline would be used, one stage for each level. With sufficient data memory bandwidth, this would increase the throughput of a packet stream by a factor of three over a single processor implementation. Alternatively, two of the levels could be combined and implemented on a pipeline stage resulting in a two stage pipeline. This might be desirable if many of the lookups terminated at a given stage (as is often the case) thus resulting in fewer lookups in later stages. Using the earlier terminology, this example contains a single flow consisting of three computationally identical tasks where differences in execution times result from the number of memory accesses associated with each of the processor stages.

A tool called *SimplePipe*[2] was used to evaluate tree level (task) to pipeline stage assignments for a problem involving 117,212 prefixes obtained from a Sprint network router (AS1239 [4, 3]). From this set, a five level tree was constructed with each successive level handling 14, 5, 3, 2 and 8 bits and the entire tree containing 504,857 nodes. Traffic was modeled as a set of 5,000 successive routing requests where the distribution of request prefixes followed those empirically obtained in [9, 2].

With a five level tree, a separate task (with its task time) may be associated with accessing each tree level. These times are given in Table 6.1. Initially, *GreedyPipe* was used to obtain the best assignment of these five tasks to processor pipelines of different lengths (1 to 16 stages). This is shown in the top graph in Figure 6.7 where, at four stages, the maximum throughput is achieved (note that task 2 takes the longest time) and remains constant after that.

Table 6.1: Task Times (μsec) for LPM, Encryption & Compression

	# Tasks	Task 1	Task 2	Task 3	Task 4	Task 5
LPM	5	2.8×10^{-2}	4.0×10^{-2}	2.6×10^{-2}	2.0×10^{-2}	1.4×10^{-2}
ENCR	11	17.4	≈ 11.4 per task for up to 10 tasks			
COMP	15	≈ 9.44 per task for up to 15 tasks				

6.2.2 AES Encryption - A Pipelined Implementation

Encryption involves transforming unsecured information (”plaintext”) into coded information (”ciphertext”) with the process being controlled by an algorithm and a key. The Advanced Encryption Standard (AES) is considered here along with the Rijndael encryption algorithm. Rijndael is an iterated block cipher which supports independently specified variable block and key lengths (128, 192 or 256 bits). The

[2] *SimplePipe* is a pipeline simulation tool based on SimpleScalar [5]. [23] An in-order processor with a clock rate of 1 GHz was modeled. All processor stage caches were taken to be of equal size with the instruction cache being sufficiently large to hold the entire stage program. A 4-way associative, 8KB data cache was assumed (No L2 cache was present) with off-chip memory latencies set at 20 processor clock cycles. The off-chip memory was taken to be structured as a set of independent banks, one for each of the processor stages where each bank holds data associated with the tasks assigned to it. A fixed stage-to-stage communications delay of 10ns was also assumed. Data for encryption and compression was obtained directly from SimpleScalar with the same parameters indicated above.

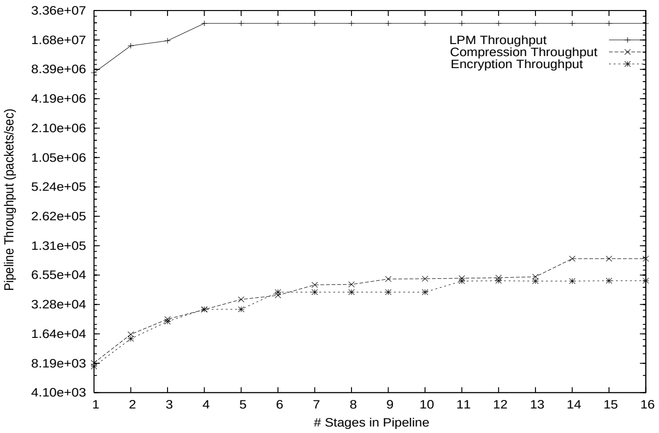


Figure 6.7: Throughput vs. Number of Stages (A single separate pipeline for each application)

algorithm has a variable number of iterations dependent on the key length. For 128 bit blocks, considered here, 10 iterations are required [1].

Algorithm transformations operate on an intermediate result called the State which is defined as a rectangular array with 4 rows and N_b columns where $N_b = block.length/32$. Transformations treat the 128-bit data block as a 4 column rectangular array of 4-byte vectors. The data block along with the 128-bit (16B) plaintext is interpreted as a State. The cipher key is also considered to be a rectangular array with four rows, the number of columns N_k being the key length divided by 32. The number of rounds (iterations) depends on the values N_b and N_k and, for 128 bit blocks, is 10.

The algorithm consists of an initial data/key addition, then 9 round transformations followed by a final round. The Key schedule expands the key entering for the cipher so that a different round key is created for each Round Transformation. Such transformations are comprised of the following four operations: *ByteSub* Transformation, a *shiftRow* Transformation, a *MixColumn* Transformation and a *Round Key Addition*. The final round is similar to earlier rounds except that the *MixColumn*(State) operation is omitted. Details associated with these operations can be found in [32]. An overview of a single processor/stage implementation is shown in on the left side of Figure 6.8 while a pipelined implementation employing loop unrolling is shown on the right side of the figure.

As indicated, each iteration or can be associated with a pipelined task. At the end of each iteration, both the schedule and the transformed packet are passed from one stage to the next. The task times associated with each of the eleven tasks are shown in Table 6.1. The values were obtained using from a SimpleScalar simulation of the algorithm. Figure 6.7 shows the throughput obtained when *GreedyPipe* is used to assign only these eleven tasks to pipelines of different lengths. Notice that since there are eleven tasks, after the pipeline length reaches a length of eleven there is no throughput improvement. Furthermore, note that given the much higher computational complexity of encryption versus LPM, the throughput that can be achieved is between two and three orders of magnitude less for pipelines of equivalent length.

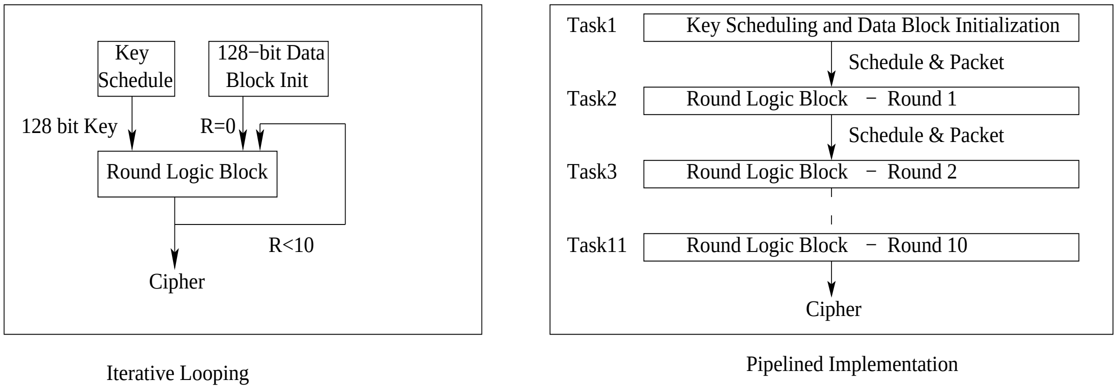


Figure 6.8: Rijndael Algorithm Implementation Block Diagram

6.2.3 Data Compression - A Pipelined Implementation

With data compression (DC) a string of characters is transformed into a new reduced length string having the same information. In this way the bandwidth required for passing a given string is reduced. *LZW* is a DC method that takes advantage of recurring patterns of strings that occur in data files. The original LZW method was created by Lempel and Ziv [46] and was further refined by Welch [42].

LZW is a "dictionary"-based compression algorithm that encodes input data by referencing a dictionary. Encoding a substring only requires that a single code number corresponding to that substring's dictionary index be written to the output file. *LZW* starts out with a dictionary of 256 characters (for the 8-bit case) and uses those as the "standard" character set. Data is then read 1 byte at a time (e.g., 'p', 'q', etc.) and encoded as the index number taken from the dictionary. When a new longer substring is encountered (say, "pq" and later say "pqr"), the dictionary is expanded to include the new substring and an index is associated with the new substring. The new index is then used when the new substring is encountered. To quickly associate substrings with indices, hashing techniques may be employed and to to limit memory size, a limit placed on maximum number of dictionary entries (say, 1024).

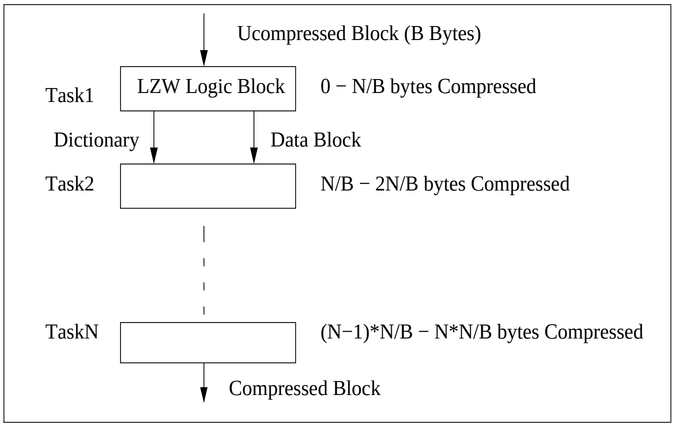


Figure 6.9: Pipelined Implementation of the *LZW* Algorithm

With a straight forward pipelined LZW implementation the input data is partitioned and successive pipeline stages operate on successive portions of the input data. Thus, if there are N stages in the pipeline, and a block of B bytes requires compression, each stage operates and compresses B/N successive input data bytes. Figure 6.9 shows the pipelined implementation. The partially compressed data block and the updated dictionary is made available to every stage of the pipeline and the resulting system throughput increases almost linearly with N.

Task times (found using Simple Scalar) associated with each task for a fourteen task implementation are shown in Table 6.1 and Figure 6.7 shows the throughput obtained when *GreedyPipe* is used to assign these tasks to pipelines of different lengths.

6.2.4 NP Example Design Results

The results shown in Figure 6.7 are for single flows executing on a single pipeline. The problem becomes more complex when:

1. Multiple flows are considered with some flows requiring multiple applications.
2. Some of the flow applications/tasks are shared and are instantiated only once.
3. Multiple processor pipelines are present with a constraint on the total number of stages available.

This more complex problem is now considered. Table 6.1, shows the execution times per task for the three applications where the output from one task becomes the input to the next task. Assume that we have the three flows indicated earlier. We assume that packets associated with Flow 1 (LPM) are 40 Bytes long while packets from Flows 2 and 3 (LPM & Encryption; LPM & Compression) are 1,500 Bytes long. Four experiments were performed:

1. **Single Pipeline:** A single 16 stage pipeline was considered where the LPM application was shared among the three flows.
2. **Two Pipelines:** Two pipelines were used where the length of each pipeline was varied between 1 and 15 stages, with the sum of the stages being set 16. Flow 1 was assigned to one pipeline and its LPM application was not shared with the other Flows. Flows 2 & 3 were assigned to the second pipeline and the LPM application was shared between them.
3. **Three Pipelines:** Three pipelines were used, one flow assigned to each, where the length of each pipeline was varied between 1 and 14 stages with the sum of the stages set to 16.

Consider the first case above where there is a single 16 stage pipeline. Entering the set of flows, their respective applications, and the application task sequence and times, *GreedPipe* performs the assignment process and obtains an assignment that maximizes the total throughput. The results in this case are that stage 1 is assigned to LPM (shared across all flows) while the remaining stages are shared by the encryption and compression components of flows 2 and 3. Table 6.2 shows the resulting best overall bandwidth (both in Gbps and Packets/second, Pps) for each flow, and the length of each pipeline for the two and three pipeline cases. Note that with a single pipeline, the bandwidth is constrained by the longest latency task which, in this case, corresponds to encryption which has the highest computational complexity.

In the second case the 16 processor stages are divided into two pipelines with their length not necessarily being equal. *GreedyPipe* was executed iteratively with a different number of stages assigned to each pipeline and, for each pipeline length selection, a near optimal assignment obtained (see Figure 6.10). For pipeline 1 (Flow 1, LPM), the results are the same as seen in Figure 6.7 since there is a single pipeline associated with that flow. As the number of stages for pipeline 1 increases however, the number remaining for pipeline 2 decreases. The result is that there are increased delays for flows 2 (LPM & Encryption) and 3 (LPM & LZW). *GreedyPipe* yields the near optimum task allocations for each of these pipeline lengths. Note that for pipeline 2, the bandwidth is dominated by the requirements of encryption. Flow 1 bandwidth is significantly improved since, having its own pipeline, its packets are now not limited by the delays for encryption.

Table 6.2: Bandwidths for Best Assignments (Pps=Packets/second; Flow 1 packet length=40 Bytes; Flow 2 & 3 packet length=1500 Bytes)

Number Pipelines	Flow 1 Rate (Gbps / Pps)	Flow 2 Rate (Gbps / Pps)	Flow 3 Rate (Gbps / Pps)	Pipe 1 Length	Pipe 2 Length	Pipe 3 Length
1	0.018/5.7x10^4	0.68/ 5.7x10^4	0.68/5.7x10^4	16	0	0
2	7.94/2.4x10^7	0.68/5.7x10^4	0.68/5.7x10^4	4	12	0
3	7.94/2.4x10^7	0.525/ 4.3x10^4	0.48/4.0x10^4	4	6	6

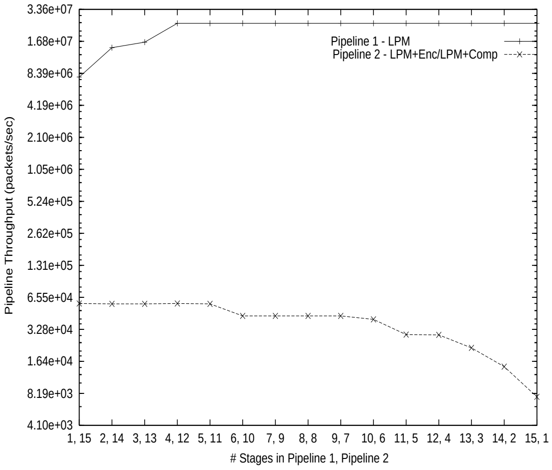


Figure 6.10: Two Pipelines - Throughput vs. Num. Stages (X, Y –> X stages for Pipe 1 & Y stages for Pipe 2)

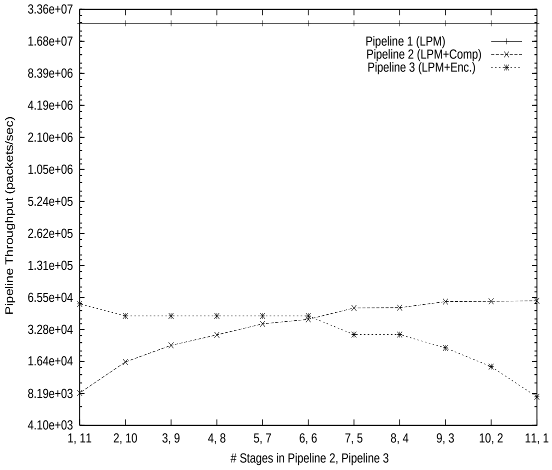


Figure 6.11: Three Pipelines - Throughput vs Num. stages for Pipelines 2 & 3; Pipeline 1=4 stages

For the third case, the 16 processor stages are shared across three pipelines with flows 1, 2 and 3 being assigned to pipelines 1, 2 and 3 respectively. Using *GreedyPipe*, all partitioning of the 16 stages across 3 pipelines were considered and evaluated. The results indicate that assigning 4 stages to Pipeline 1 (Flow 1, LPM) achieves the highest throughput. Figure 6.11 plots results associated with pipeline 1's length equal to 4 stages. As more of the remaining 12 stages stages are assigned to flow 2 there are fewer for flow 3. Given the pipelined implementations of encryption and compression, the crossover point on the graph corresponds to the highest throughput result (also see Table 6.2. Due to the fact that the length of pipelines 2 and 3 are constrained, the full effect of pipelining the flows 2 and 3 cannot be realized and the maximum throughput for those flows is less than the two pipeline case.

6.2.5 Simulated Annealing for Task & Memory Assignment

The simulated annealing approach was applied to the pipelined implementation of the AES Encryption algorithm which involved a single flow with 11 tasks. This experiment was primarily conducted to explore the effect on performance/throughput with the variation in number of stages and the number of memory modules available for the stages. As described earlier, each iteration of the encryption process can be associated with a pipelined task. The task times associated with each of the eleven tasks are shown in Table 6.3. These values were obtained using SimpleScalar with the times modified to reflect an environment where there is no cache, no memory contention between tasks, and memory access time set to 15 clock cycles.

Table 6.3: Task Times (μsec) for AES Encryption assuming 15 clock cycles per memory access

Appl:# Tasks	Task 1	Task 2 to Task 11
ENCRYPTION : 11	76.5	$\approx$ 24.4 per task for up to 10 tasks

The data inputs for each of the tasks are the key for the corresponding round and the partially encrypted 128-bit text. We assume that there are buffers available for communication of data between the processors (e.g. next neighbor registers in Intel IXP 2400). Thus the partially encrypted text can be assumed to be communicated between the processors using these buffers. The round keys, as described earlier are generated by the first task and stored in the SRAM/DRAM for use by the remaining tasks. Thus each task (except $task_1$) needs to access the memory to get the corresponding round key and then execute a round of encryption on it. We assume that the memory width is 32-bits, so that it takes 4 memory requests to load or store the 128-bit key. Since the first task generates all the 10 keys for the 10 rounds and stores them in the memory, it would involve $10 \times 4 = 40$ memory accesses. Since the remaining 10 tasks correspond to the 10 rounds, each of them would involve reading the 128-bit key once, which is equivalent to 1×4 memory accesses.

Although, this experiment was conducted with an assumption that the first pipeline stage (key generation stage) has access to all the memory modules, our implementation of the simulated annealing algorithm is more constrained and does not permit multiple memory modules to be accessed by one stage. Hence, the results presented for the experiment are approximate values and the actual throughput for these experiments is likely to be lower due to increased memory contention.

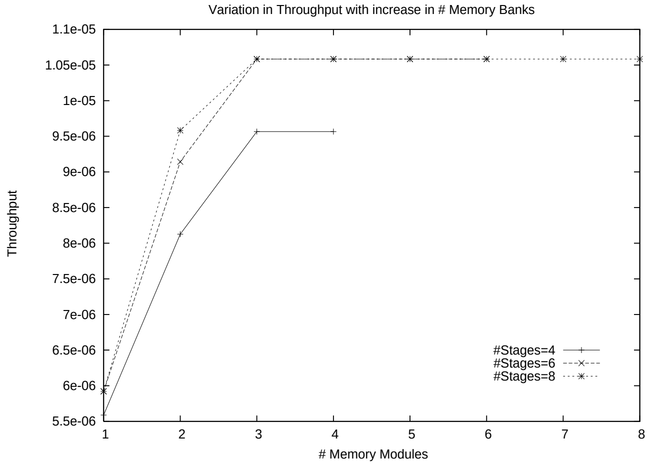


Figure 6.12: Throughput vs Number Memory Banks

Figure 6.12 shows the variation in throughput with an increase in the number of memory modules available. Each point on a curve represents the best throughput possible for the corresponding number of memory modules. The best throughput value was obtained by evaluating (using the simulated annealing algorithm) all the allocation permutations associated with memories and processors. Experiments were conducted for varying number of stages in the pipeline (viz. 4, 6 and 8). As expected, for a given number of stages, the throughput increases as the number of memory modules available is increased since with an increased number of modules there will be a decrease in memory contention delays. The throughput does not improve beyond a certain point(Number of Stages=8), since that is the maximum possible throughput corresponding to the maximum delay task ($task_1$).

This chapter demonstrated the use of task scheduling algorithms to evaluate the effects of pipeline depth, task sharing, task partitioning and memory to processor allocation on the performance on a multi-processor pipelined system. The next chapter presents a performance comparison of the task scheduling algorithms discussed in this thesis.

Chapter 7

Comparative Study of Solution Approaches

This chapter presents a comparison of the four solution approaches described earlier in the thesis: Complete Enumeration (CE), *GreedyPipe*, Dynamic Programming and Simulated Annealing. These approaches will be examined in terms of their complexity, execution time, applicability and the accuracy of the solutions obtained.

7.1 Complexity Analysis

Regarding the efficiency of algorithms, a standard measure is the number of elementary computer operations required to solve the problem or the complexity of the solution approach in the worst case. Average performance is generally not a safe measure since there may be particular cases that behave much worse than the average. Of primary interest is the comparison of the various solution approaches, as a function of the problem size where the size parameter(s) selected is/are based on the given problem domain. The time complexity of all the four algorithmic approaches was discussed earlier in the thesis and is being reproduced here for convenience. We will consider three classes of problems:

- **S**ingle **P**ipeline, **M**ultiple **F**low (**SP/MF**)

- **S**ingle **P**ipeline, **M**ultiple **F**low, **S**hared **T**asks (**SP/MF/ST**)
- **S**ingle **P**ipeline, **M**ultiple **F**lows, **S**hared **M**emory Modules(**SP/MF/SM**)

Table 7.1 shows the complexity for all the algorithm approaches for the above mentioned classes of problems. The time complexity derivation for each algorithm was presented in the chapters describing the algorithms, earlier in the thesis. In the table, N is the number of flows in the system, M is the number of tasks in each flow and R is the number of stages in the pipeline.

Table 7.1: Time Complexities

	SP/MF	**SP/MF/ST**	**SP/MF/SM**
CE: Complete Enumeration (Section 2.5)	$O((\frac{(M+R)!}{M!R!})N)$	$O((\frac{(M+R)!}{M!R!})^N)$	$O((\frac{(M+R)!}{M!R!})^N(R^2))$
GreedyPipe (Section 3.2)	$O(NR)$	$O(2^N R)$	
DP: Dynamic Programming (Section 4.2)	$O(M^2RN)$	$O(M^{2N}R)$	
SA: Simulated Annealing (Section 5.2)	$O((R^2NM)^2)$	$O((R^2NM)^2)$	$O(R^2)(R^2NM)^2)$

From Table 7.1, *GreedyPipe* has a polynomial time complexity irrespective of the size or class of the problem. The rate of growth of execution time of CE may be comparable to that of DP or *GreedyPipe* for smaller problems of the SP/MF class. However, as the problem size increases, the time complexity of CE method increases rapidly due to the factorial component and attains a very high value for all classes of problems. The performance of DP is comparable to *GreedyPipe* for the SP/MF class of problems, but it's complexity also increases exponentially with N for the SP/MF/ST class of problems. SA has a polynomial time complexity for all classes of problems but being a statistical approach, irrespective of the problem size, it's complexity is very high. Comparing the four approaches, *Greedypipe* involves minimum growth in execution time for all classes of problems. DP is an acceptable option for the SP/MF class of problems but may become infeasible for larger problems of the SP/MF/ST class. SA is applicable for the SP/MF/SM class of problems and is a preferable approach to the complete enumeration method for larger problems of that type.

7.2 Execution Time and Accuracy

An alternative way to examine the computational complexity of the algorithms is to compare the actual execution time of the algorithms for the same set of problems. However, there is another element of concern when evaluating the performance of the various solution approaches and that is, how closely do the results obtained using these methods match with the true optimal results. The following subsections illustrate the difference in performance of the of the algorithms for the following five classes of problems :

- **S**ingle **P**ipeline, **S**ingle **F**low (**SP/SF**)
- **S**ingle **P**ipeline, **M**ultiple **F**low (**SP/MF**)
- **S**ingle **P**ipeline, **M**ultiple **F**low, **S**hared **T**asks (**SP/MF/ST**)
- **S**ingle **P**ipeline, **S**ingle **F**low, **S**hared **M**emory Modules(**SP/SF/SM**)
- **S**ingle **P**ipeline, **M**ultiple **F**lows, **S**hared **M**emory Modules(**SP/MF/SM**)

Figures illustrating the results of a number of experiments are provided in the following subsections for a comparison of the correctness of the solutions obtained using the four algorithm approaches and the execution time required by the four algorithm approaches. Each data point presented represents the results of averaging 10 experiments. Each of the four algorithms, CE, *Greedypipe*, DP and SA, was used to generate the task to pipeline assignment for every experiment. The number of tasks in a flow for each experiment were equal to twice the number of stages in the system and the task times were randomly selected from a uniform distribution ranging from 0 to 10 floating point time units. For the experiments of the classes SP/SF, SP/MF and

SP/MF/ST, an independent memory module was made available to each processor, to avoid any memory contention. The experiments examine the effect of increase in number of stages and tasks on the throughput and the execution time of the algorithm implementations. All the experiments were conducted on a 1.4GHz Intel processor system.

7.2.1 SP/SF

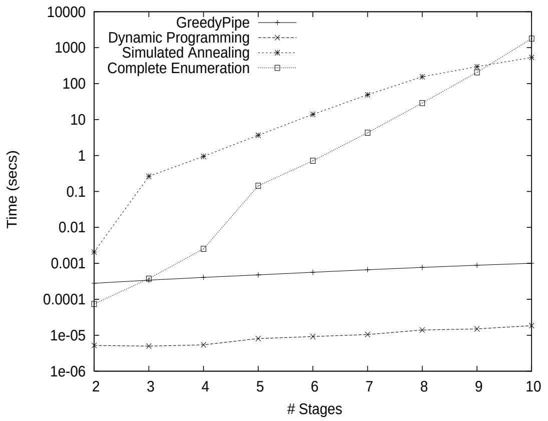


Figure 7.1: Execution Time, Single Flow

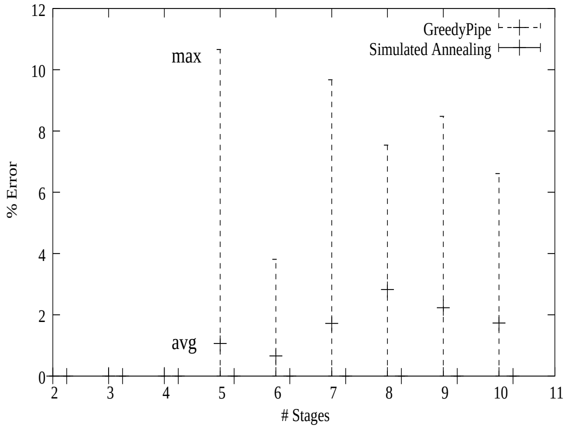


Figure 7.2: % Error in Throughput, Single Flow (SA is optimal over all experiments)

Figure 7.1 illustrates the results for a system with a single flow on a log-scale. As expected, the execution times for DP and *GreedyPipe* both increase linearly with the increase in the number of stages. The execution time for *GreedyPipe* is more than DP because of the difference in constant time factor due to traversal in both forward and reverse directions, data structure maintenance etc. As described in Chapter 3, *GreedyPipe* involves maximizing the throughput across all the stages. The data structure maintenance involved in *Greedypipe* implementation was a little more complex to add a constant time to the execution time and avoid comparisons across all the stages. As expected, execution time for CE increases rapidly with the number of stages and tasks. The execution time for SA also increases with the number of stages (as expected from Table 7.1) and is the maximum of all the approaches for most of the experiments. Figure 7.2 shows the percentage error for throughput results obtained by different algorithm approaches. Since DP produces optimal solutions, the throughput obtained by DP for each experiment is the same as that obtained using CE. The heuristic approach produces optimal results when the number of stages are few however, beyond 4 stages, the results obtained by it deviate from the optimal. The maximum deviation from the optimal is seen at number of stages equal to 5, however, the average deviation for *GreedyPipe* is less than 5% of the optimal. Simulated annealing produces an optimal allocation for all the experiments and thus there are no entries on the figure.

As seen from the experiments, for a single flow system with no task sharing and no memory contention, DP is the most efficient approach in terms of both execution time and accuracy of the solutions obtained using it.

7.2.2 SP/MF

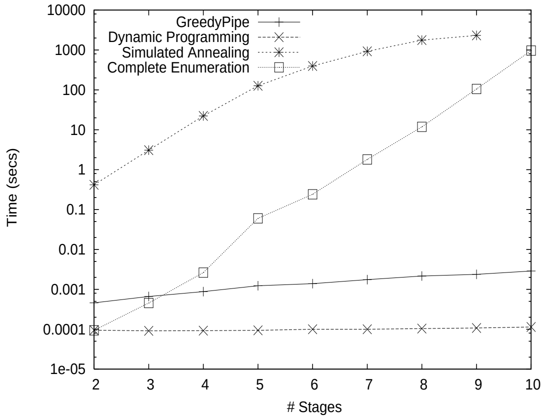


Figure 7.3: Execution Time for Multiple Flows.

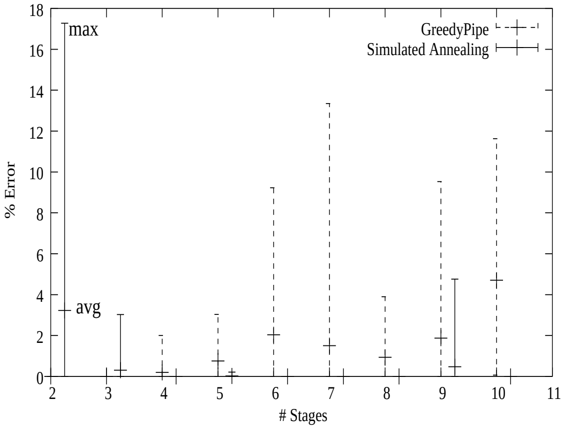


Figure 7.4: % Error in Throughput, Multiple Flows.

For a system with multiple flows and no shared tasks, each flow can be treated independently. It is equivalent to having multiple SP/SF problems and thus, a linear increase in execution time should be expected with the increase in number of flows in the system. Figure 7.3 shows the results of the experiments conducted on a system with 3 flows. The results seem similar to those in Figure 7.1 except for the relative increase in the execution time. There is a linear increase in execution times due to increase in the number of flows, however it is not as apparent due to the log-scale plot. From Figure 7.4, DP, as before, always obtains an optimal assignment of tasks to pipeline. *GreedyPipe* produces less optimal results as the number of stages increases and the maximum error for *GreedyPipe* is around 13% (at number of stages=7) of the opimal. However, the average percentage error is still within 5% (as in case of SP/SF). SA also produces unoptimal results with a maximum error percentage within 5% (at number of stages=9) of the optimal value while the average error for solutions obtained using SA is within 2% of the optimal.

For a multiple flow system with no task sharing and no memory contention, again DP is the most efficient approach in terms of both execution time and accuracy of the solutions obtained using it.

7.2.3 SP/MF/ST

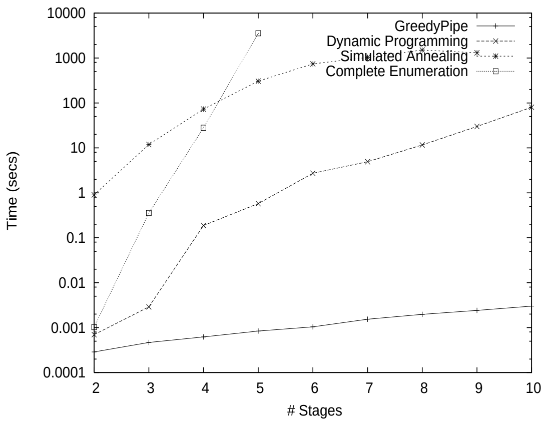


Figure 7.5: Execution Time for Multiple Flows, Shared Tasks

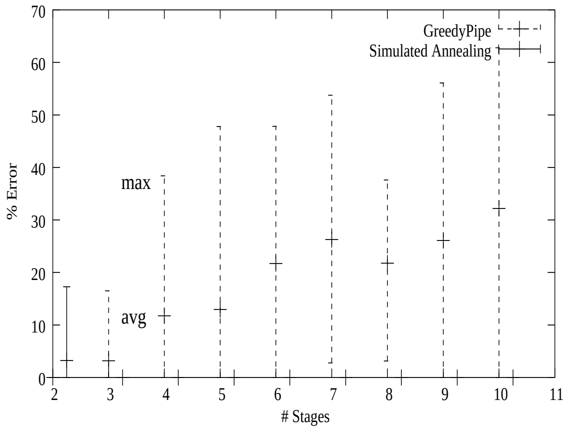


Figure 7.6: % Error in Throughput, Multiple Flows, Shared Tasks

Figure 7.5 presents the results for a system with 3 flows and tasks shared between the flows such that, 25% of the tasks in each flow are common with the other flows. As the figure shows, there is not a significant difference in the performance of both the SA and the *GreedyPipe* methods. When SA is applied to a system with shared tasks, the only additional functionality for SA is to validate the task allocation to comply with the sharing constraint and thus it's performance is not as much affected. *GreedyPipe* performance would be further affected with the increase in number of flows N, since from Table 7.1, it's execution time reduces by a factor of 2^N. Even then, it's performance in terms of execution time is still expected to be much better than that of DP and CE. Due to the interdependence between flows, CE and DP cannot treat the flows independently and thus, as seen in the figure, the execution time for a certain number of stages in a system is much higher than the corresponding execution time (Figure 7.3) for a SP/MF system. The performance for both DP and CE is expected (from complexity analysis in the previous section) to deteriorate significantly with the increase in number of flows. For DP, as the number of flows and tasks in flow increase, the space complexity also increases by the same order and makes it less feasible for use. Figures 7.5 and 7.6 show the data points for CE only up to 5 stages since beyond that, it will take days or even weeks to compute the optimal assignment using the complete enumeration method. Figure 7.6 shows the percentage error in the throughputs obtained when the four algorithmic approaches are applied to the same set of experiments. From the figure, SA obtains an optimal allocation for most of the experiments. It gives non-optimal results at number of stages=2, with the average error less than 5% and the maximum error around 17%. The throughput values obtained using *GreedyPipe* become less optimal as the number of stages and the number of tasks per flow are increased. While up to 5 stages and 10 tasks per flow, the average percentage error is around 10%, it goes up to 30% when the number of stages equals 10 and the number of tasks in each flow is equal to 20. Also the maximum percentage error for *GreedyPipe* goes to almost 60% of the optimal at number of stages=10.

The applicability of an algorithm for a system with shared tasks will depending on the size and complexity of the problem, required accuracy of the solution and the allowed run-time delays for the algorithm implementations.

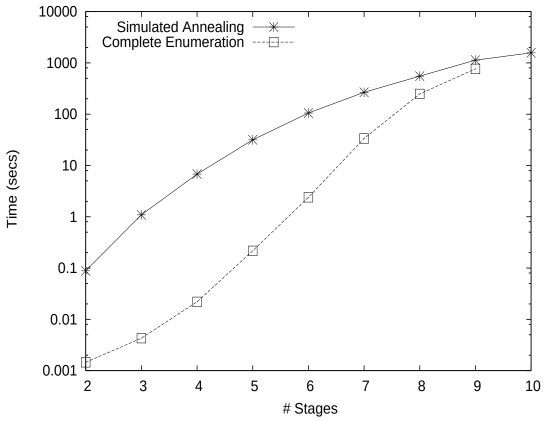


Figure 7.7: Execution Time; Single Flow, Memory Contention

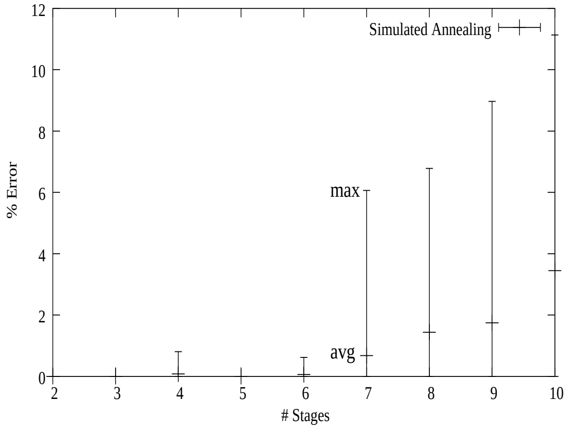


Figure 7.8: % Error in Throughput, Single Flow, Memory Contention

7.2.4 SP/SF/SM

This section examines the algorithm performance on a system with a single flow but the number of memory modules in the system is less than the number of processors so that memory contention also needs to be considered when the tasks are allocated to the pipelined processors. For all the experiments, each memory module was shared by a maximum of three processors and the memory modules were randomly allocated to the processors. Each memory contention computation requires a complete layout of the tasks allocation. Since both DP and *GreedyPipe* work on one stage at a time and propagate the results to the subsequent stages to get the final task allocation, they cannot be applied to systems where memory contention needs to be considered. Figure 7.7 shows the execution times for both SA and CE. For systems with up to 8 stages, the run-time for CE is less than that of SA, however with more than 8 stages, the execution times for CE become comparable to that of SA. Figure 7.8 shows the throughputs obtained using CE and SA. As the figure shows, the allocations produced by SA are very close to optimal for less number of stages. The maximum percentage error in throughput for allocations obtained using SA is around 11% at number of stages=10 and number of tasks=20. The average error in throughput for SA is less than 5%. From the figure, for stages more than 10, CE may not perform as well as SA and using SA would be much more practical.

7.2.5 SP/MF/SM

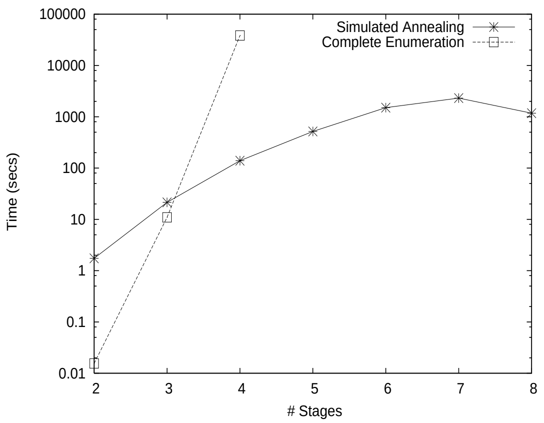


Figure 7.9: Execution Time; Multiple Flows, Memory Contention

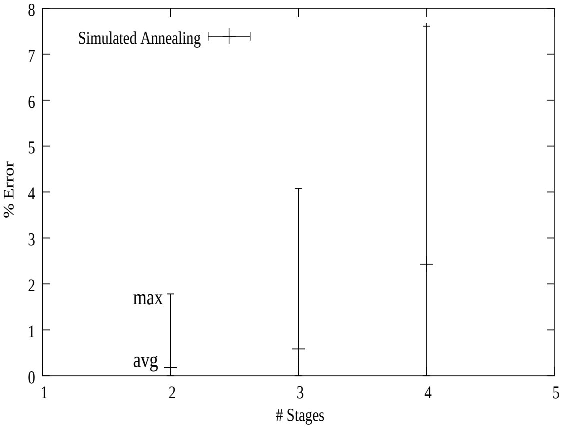


Figure 7.10: % Error in Throughput; Multiple Flows, Memory Contention

This section presents a performance comparison of the algorithms for systems with 3 flows and no shared tasks such that again memory contention needs to be considered when allocating the tasks to the processor pipeline. For all the experiments, each memory module was shared by a maximum of three processors and the memory modules were randomly allocated to the processors. For such a system, the flows cannot be treated independently, since the maximum memory contention computation requires the complete task allocation for all the flows (explained in Appendix A). Thus the execution time for CE grows exponentially with increase in number of stages and tasks as shown in Figure 7.9. The figure shows values only up to 4 stages since it was practically impossible to determine the allocation using CE due to it's very high execution time. As seen from the figure, the execution time for SA is still feasible for even higher number of stages. Also, from Figure 7.10, up to 4 stages, the results obtained using SA are very close to optimal with the maximum error within 8% of the optimal. Thus, beyond 4 stages, SA is the only feasible algorithm that can be applied to the systems of the class SP/SF/SM.

Different algorithm approaches will present effective solutions for different system scenarios. The experiments show that the choice of an algorithm for a problem will depend on the complexity and size of the problem, the expected accuracy of the solution obtained using the algorithm, and the available run-time for the algorithm. Also, the figures in the above sections represent the relative performance of the algorithms and the actual execution times could be further improved with optimized algorithm implementations.

Chapter 8

Conclusions and Future Work

8.1 Contributions

With the increasing availability of chip multi-processors (CMPs), the problems associated with assigning tasks to processors in a manner that maximizes performance will gain in importance. In selected CMPs tailored towards the networking environment, single processor pipelines can be configured under software control. Network Processors have this capability and this thesis has examined the problem of task to pipeline stage assignment in this domain. A formal definition of the task scheduling problem was presented including the constraints associated with it. Additionally, the thesis also described a resource (memory in this case) contention model and used it to formulate the performance metrics for the scheduling problem. Obtaining optimal assignments is an NP hard problem and exhaustive search techniques can take multiple days to find an optimal assignment for reasonable sized problems. The thesis described three optimization algorithms based on heuristics (*GreedyPipe*), dynamic programming and a statistical approach (Simulated Annealing) to quickly perform optimal or nearly optimal task-to-pipeline stage assignments in a multiple flow, multiple pipeline environment. In addition to presenting details associated with the design and performance of the algorithms, the thesis also illustrated how the algorithms can be together employed as a design tool in situations where the effects of pipeline depth, task sharing, task partitioning, number of pipelines and memory to processor assignment are to be explored as part of the design process. Initially, the algorithms were used for a generic case employing synthetic data and subsequently they were applied to a specific NP problem where the applications include routing, compression and encryption. The thesis showed how the algorithms can be used to obtain good assignments in an environment where the total number of processor stages in all the pipelines is constrained (e.g., 16). This is similar to what is available in the Intel IXP2800. Finally, the thesis presented a comparison of the four solution approaches, Complete Enumeration (CE), *GreedyPipe*, Dynamic Programming and Simulated Annealing in terms of their complexity, execution time, applicability and the accuracy of the solutions obtained using them.

8.2 Future Work

Most of the recent Network Processor models are multi-threaded multiprocessor systems. The scheduling algorithms and the memory model, however are constrained to single threaded systems. Further work can be done to formulate the problem including changing the memory contention model for a multi-threaded system and adapt the optimization algorithms accordingly.

Secondly, the algorithms as of now may find an off-line use, so that the tasks and flows are allocated to processor pipelines as part of the initialization process for the system. Thus, any variation in the traffic will need to be examined and the tasks will need to be reassigned offline to adapt to the changes in traffic. Future work could involve extension to the algorithms so that the traffic updates can be passed as feedback to the algorithms. Thus, the algorithms on receiving a feedback could reallocate the tasks to pipelined processors on a real-time basis. Considerable improvement in performance may be required for the task scheduling to be done on a real-time basis, however, precomputation of sets of likely conditions that required reassignment can be stored along with associated optimal assignments and potentially used for fast task assignment.

Appendix A

Memory Contention Delay

We determine an expression to calculate the worst case memory delay for each processor, for the memory contention model described in Chapter 2. The total number of memory accesses by all the tasks of flow j allocated to processor k is represented by mem_{jk} so that

$$mem_{jk} = \sum_{i=1}^{M_j} X_{ijk} m_{ij} \tag{A.1}$$

For a single flow, the subscript j can be dropped.

Say we need to compute the maximum possible contention for tasks of flow j' on processor k'. For a case with a single flow, the number of contending memory accesses from a processor k where $k \neq k'$, is given by mem_{1k} (single flow : $j' = 1$). However, in case of multiple flows, the number of contending memory accesses from a processor k is given by the maximum value of mem_{jk} across all flows (for $1 \leq j \leq N$), since packets of different flows could be executing on different processors. This can be further explained by the example shown in Figure A.1. The example shows two cases of a problem with two flows, F_1 and F_2. For simplicity, each flow is assumed to have 3 tasks with each task allocated to a different processor. The contending memory accesses for the *current packet* in any processor k is represented by mem_k.

Figure A.1 (a) shows a case where a packet of flow F_1 is being processed in each of the stages. Thus, $mem_1 = mem_{11} = 2$, $mem_2 = mem_{12} = 3$ and $mem_3 = mem_{13} = 0$.

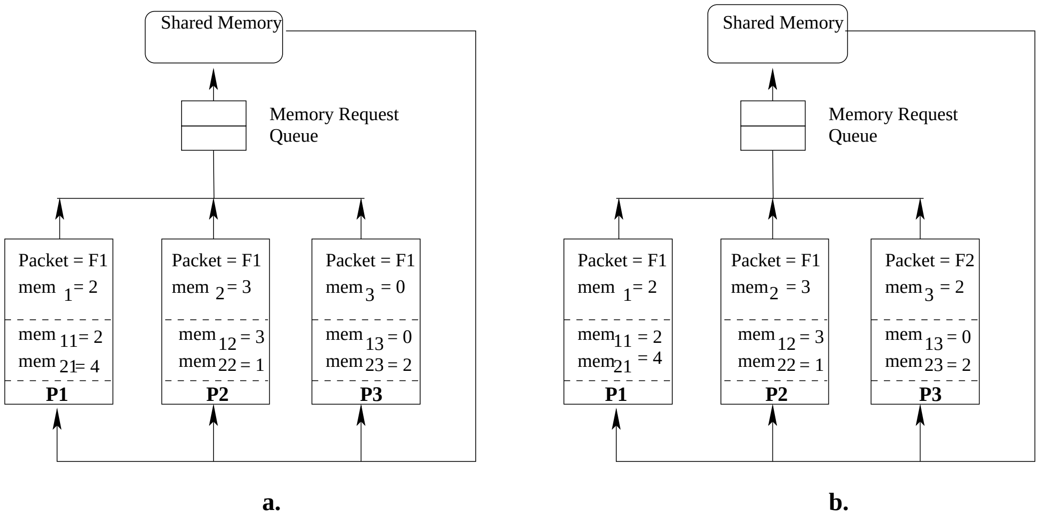


Figure A.1: Multi-flow Memory Contention

Now, we consider Figure A.1 (b), which is a variation of the previous case except that now processor P_3 has a packet of flow F_2 being processed in it. Accordingly, the number of contending memory accesses from processor P_3 is given by $mem_3 = mem_{23} = 2$. Hence, the contending memory accesses by a processor vary depending on the type of packet being processed in it. Thus, for multiple flows, the number of contending memory accesses by the tasks allocated to processor k is given by:

$$mem_k = \max_{j=1}^{N}[mem_{jk}] \tag{A.2}$$

Accordingly, the number of contending memory accesses for the processors in Figure A.1 are $P_1 : mem_{21} = 4$, $P_2 : mem_{12} = 3$ and $P_3 : mem_{13} = 2$.

Now we will calculate the *total* number of contending memory accesses when a packet of flow j' is being executed on processor k'. The number of memory accesses required by processor k' to process a packet of flow j' is represented by $mem_{j'k'}$. While evaluating the contending memory accesses due to a processor k where $k \neq k'$, though the number of memory accesses made by the processor (by Equation A.2) would be mem_k, we need to find just how many of those actually contend with $mem_{j'k'}$.

Assuming a worst case scenario, if $mem_{j'k'} > mem_k$, the number of contending memory accesses by processor k is equal to mem_k, since in the worst case everytime processor k' makes a request, a request by processor k is being serviced. Thus, processor k' will have to wait for one memory request for each memory request it makes till it has completed a total of mem_k memory requests. By using a similar argument, for the same case, if $mem_{j'k'} \leq mem_k$, the number of contending memory accesses by processor k is equal to mem_{jk}. This is illustrated in the example shown in Figure 2.1. where it shows the computation of the total number of contending memory accesses for P_1 (mem_1). Since, $mem_3 < mem_1$, the number of contending memory accesses from $P_3 = mem_3 = 1$. Also, since $mem_1 < mem_2$, the number of contending memory accesses from $P_2 = mem_1 = 2$.

Thus, the maximum number of contending memory accesses, c_{jk}, for a packet of flow j requiring mem_{jk} memory accesses on processor k is given by :

$$c_{jk} = \sum_{p=1, p \neq k}^{R} z_p \qquad \text{where } z_p = \begin{cases} mem_p & \text{if } mem_p < mem_{jk} \\ mem_{jk} & \text{if } mem_p \geq mem_{jk} \end{cases} \tag{A.3}$$

For multiple memory modules, with the processor-memory block association represented by A_{km}(as described in Section 2.1), z_p from Equation A.3 should be considered only if both processor k and processor p access the same memory block. Thus, for a system with multiple memory modules and R processors, c_{jk} is given by:

$$c_{jk} = \sum_{p=1}^{R} z_p A_{pm} \qquad \text{where } A_{km} = 1 \tag{A.4}$$

In Equation A.4, $A_{km} = 1$ implies that the processor k accesses memory block m. Since A_{pm} can either be 0 or 1, z_p gets added to the total contending accesses (c_{jk}) only if $A_{pm} = 1$ i.e. if processor p accesses memory module m.

Appendix B

Phoenix Toolset Manual (ver 1.0)

The *Phoenix* toolset is a set of algorithms to allocate sequential tasks optimally, to a processor pipeline. It can be configured to use a heuristic approach (*GreedyPipe*), dynamic programming or a statistical approach (*Simulated Annealing*). The tool has been written using the C programming language. The tool is enabled to accept the input parameters at the command prompt or in the form of a configuration file. *Phoenix* has been tested on both Solaris and Linux operating systems. To obtain the *Phoenix* Toolset, e-mail your requests to seema/jbf@ccrc.wustl.edu

B.1 Usage

The following command needs to be issued at the command prompt to run the *Phoenix* tool:

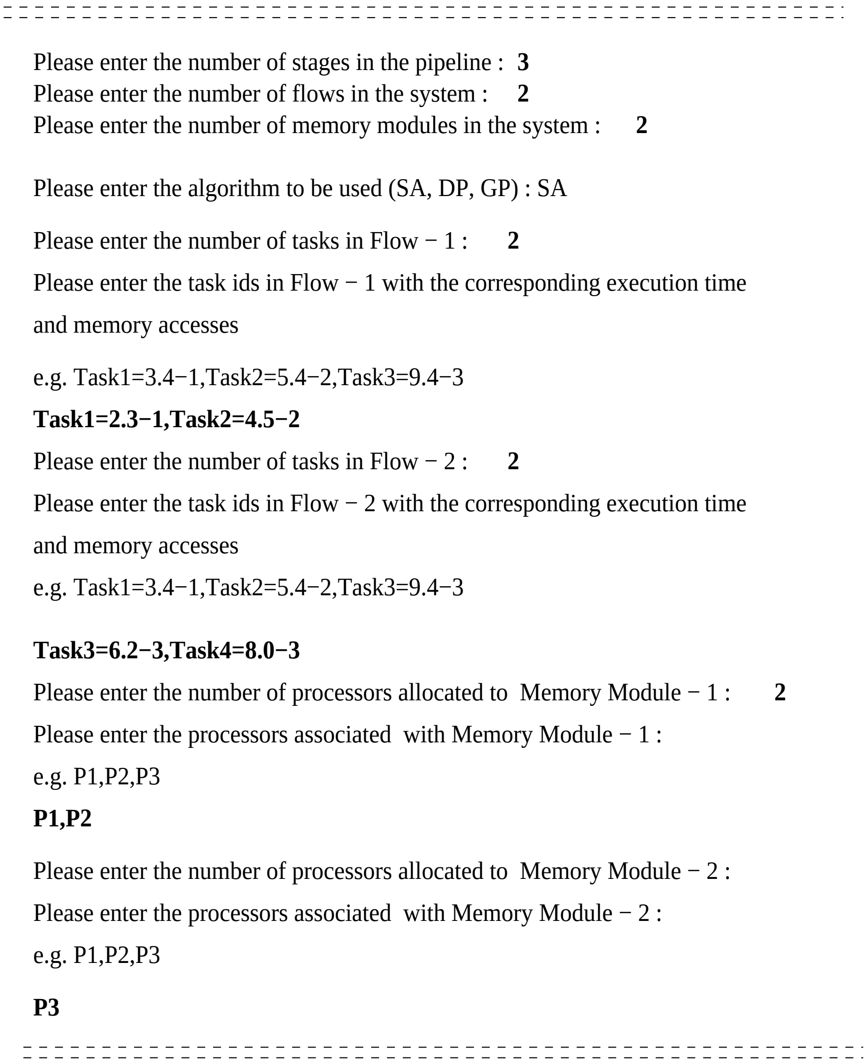
```
$ phoenix
```

The toolset accepts the following command line arguments:

Table B.1: Sample - Interactive Command Line Option

default	prints the help message
-f	reads the parameters from a configuration file named *config.txt*
-c	gets the input parameters from the command line

Table B.2: Command Line Options For *Phoenix*



```
Please enter the number of stages in the pipeline :  3
Please enter the number of flows in the system :     2
Please enter the number of memory modules in the system :      2

Please enter the algorithm to be used (SA, DP, GP) : SA

Please enter the number of tasks in Flow − 1 :      2
Please enter the task ids in Flow − 1 with the corresponding execution time
and memory accesses

e.g. Task1=3.4−1,Task2=5.4−2,Task3=9.4−3
Task1=2.3−1,Task2=4.5−2
Please enter the number of tasks in Flow − 2 :      2
Please enter the task ids in Flow − 2 with the corresponding execution time
and memory accesses
e.g. Task1=3.4−1,Task2=5.4−2,Task3=9.4−3

Task3=6.2−3,Task4=8.0−3
Please enter the number of processors allocated to  Memory Module − 1 :       2
Please enter the processors associated  with Memory Module − 1 :
e.g. P1,P2,P3
P1,P2
Please enter the number of processors allocated to  Memory Module − 2 :
Please enter the processors associated  with Memory Module − 2 :
e.g. P1,P2,P3

P3
```

B.1.1 Command Line Format

The input parameters to the toolset can be provided through the command line by using the following command:

$ phoenix -c

On issuing the above command, the user is prompted to enter the system configuration interactively as shown in Figure B.1. The entries in **bold** are the user entered values. The tool expects an integer value to be entered for the number of processor stages, the number of flows , the number of memory modules and the number of tasks in flows. The minimum integer value for all the entities can be 1. The tool also requires to know the algorithm to be used. The algorithm options are :

- GP : *GreedyPipe*
- DP : Dynamic Programming
- SA : Simulated Annealing
- CE : Complete Enumeration

For CE, if the problem size is large, the tool gives a warning about high execution time and prompts for confirmation of the algorithm to be used. If a memory module is shared between processors, the tool ignores the user entered option for the algorithm to be used and uses SA. The task details pertaining to each flow are entered in terms of task Ids, their respective execution times and the total number of memory accesses required by each task as

$TaskId_1$ = *Execution Time*$_1$-*Memory Accesses*$_1$, $TaskId_2$ = *Execution Time*$_2$-*Memory Accesses*2,....

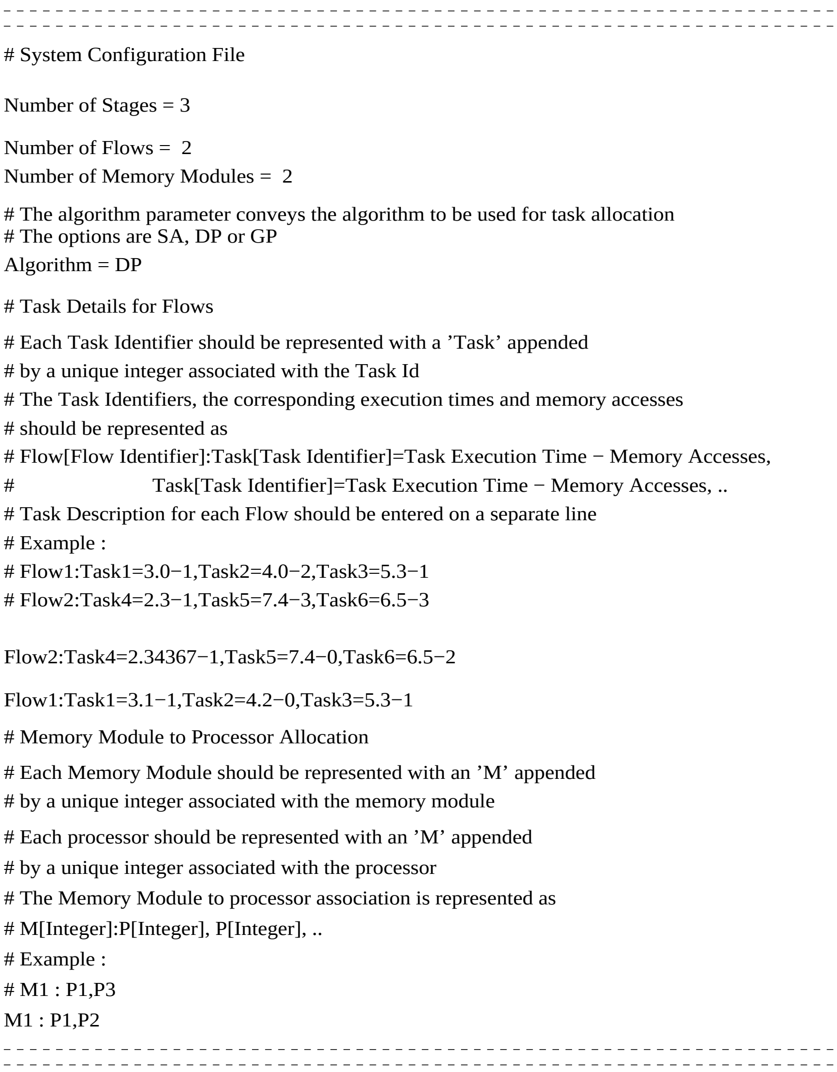
The task Ids need to have the format *Task*{*xxx*} where 'xxx' is a unique integer identifying the task. The task Id can also be represented using a short-hand notation as *T*{*xxx*}. The execution time could be any decimal number to a precision of 6 decimal places. The order of the entered tasks also implies the sequence in which the tasks need to be executed. $MemoryAccesses$ is the total number of memory accesses required by the task and should be an integer entry.

The memory to processor allocation details are entered in terms of the memory module number and the processors associated with it.

$MemoryModule_1 = \{Processor_1,\ Processor_2,....\}$

The memory modules need to have the format *M*{*xxx*} where 'xxx' is a unique integer identifying the memory module. The processors need to be represented as *P*{*xxx*} where 'xxx' is a unique integer identifying the processor. After getting all the system configuration parameters, the tool computes the optimal allocation and displays the task assignments for the pipeline. All the entries on the command line are case insensitive.

B.1.2 Configuration File Format



```
# System Configuration File

Number of Stages = 3

Number of Flows =  2
Number of Memory Modules =  2

# The algorithm parameter conveys the algorithm to be used for task allocation
# The options are SA, DP or GP
Algorithm = DP

# Task Details for Flows

# Each Task Identifier should be represented with a 'Task' appended
# by a unique integer associated with the Task Id
# The Task Identifiers, the corresponding execution times and memory accesses
# should be represented as
# Flow[Flow Identifier]:Task[Task Identifier]=Task Execution Time − Memory Accesses,
#                       Task[Task Identifier]=Task Execution Time − Memory Accesses, ..
# Task Description for each Flow should be entered on a separate line
# Example :
# Flow1:Task1=3.0−1,Task2=4.0−2,Task3=5.3−1
# Flow2:Task4=2.3−1,Task5=7.4−3,Task6=6.5−3

Flow2:Task4=2.34367−1,Task5=7.4−0,Task6=6.5−2

Flow1:Task1=3.1−1,Task2=4.2−0,Task3=5.3−1

# Memory Module to Processor Allocation

# Each Memory Module should be represented with an 'M' appended
# by a unique integer associated with the memory module

# Each processor should be represented with an 'M' appended
# by a unique integer associated with the processor
# The Memory Module to processor association is represented as
# M[Integer]:P[Integer], P[Integer], ..
# Example :
# M1 : P1,P3
M1 : P1,P2
```

Figure B.1: Sample - *Phoenix* Toolset Configuration File

The input parameters to the tool can alternatively be provided through a configuration file, *config.txt*, by executing the following command

```
$ phoenix -f
```

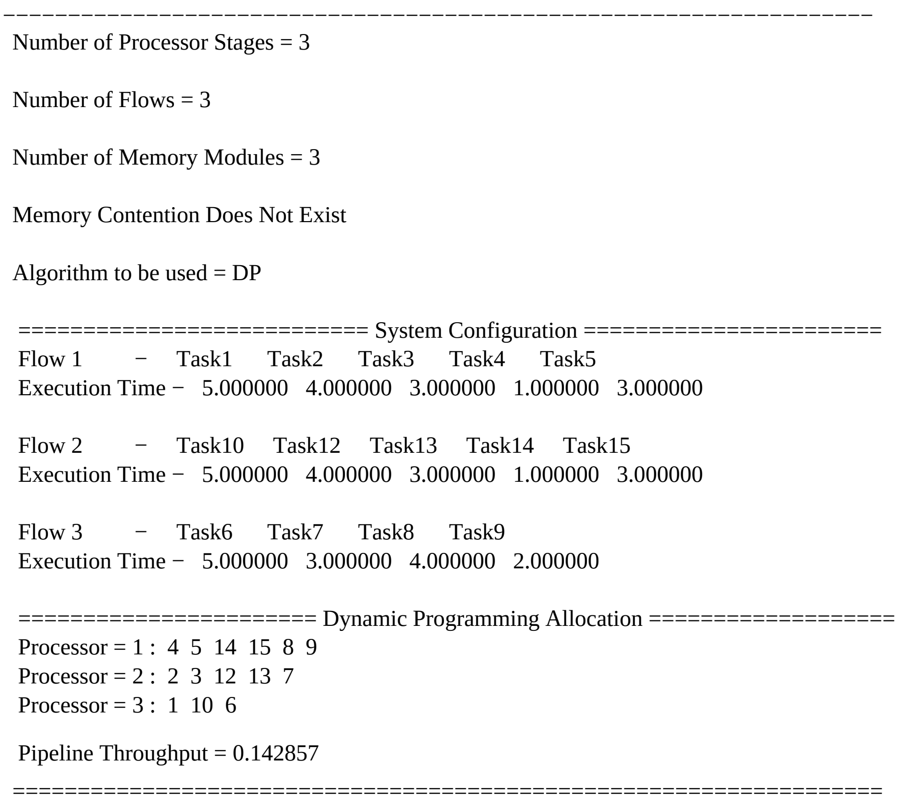
The configuration file should be placed in the same directory as the toolset executable. The configuration file contains the tool and system configuration details like the number of processor stages, the number of flows in the system, the number of memory modules, the task details for each of th flows etc. Figure B.1 shows a sample format of the configuration file. The contents of the file are not case sensitive. The lines in the file starting with the character '#' are considered to be comments. The system description primarily consists of the following entries.

- *Number of Stages* : Denotes the number of processor stages in the pipeline and expects an integer value to be entered after the '=' delimiter.
- *Number of Flows* : Specifies the number of flow classifications in the system and requires an integer index to be entered following the '=' delimiter.
- *Number of Memory Modules* : Specifies the number of memory blocks in the system and needs an integer index to be entered following the '=' delimiter.
- *Algorithm* : Specifies the algorithm to be used for task scheduling. The valid options are SA, DP, GP or CE (described in the previous subsection).
- *Task Details for Flows* : This set of entries, specifies the task details for each of the flows. Each entry depicting the details for each flow should be entered on a new line and should comply with the following format.

  FlowId$_1$*:TaskId*$_1$*=Execution Time*$_1$*-Memory Accesses*$_1$, *TaskId*$_2$*=Execution Time*$_2$*-Memory Accesses*$_2$*,..*

  The flow identification needs to be entered in the format *Flow*{*xxx*}, where 'xxx' represents the unique integer identifying the flow. The flow Id can also be represented using a short-hand notation as *F*{*xxx*}. The flow identification The sequence of task entries in a flow entry represents the order of execution of the tasks.

- *Memory Module to Processor Allocation* : This set of entries specifies the memory to processor allocation. The memory module identification should be entered in the format $M\{xxx\}$, where 'xxx' represents the unique integer identifying the memory module. The memory module identifier is followed by the set of processors associated with the memory module. A processor is represented as $P\{xxx\}$, where 'xxx' indicates the unique integer identifying the processor. The number of processors allocated to a memory module indicates if there is any memory contention.



```
----------------------------------------------------------------------
 Number of Processor Stages = 3

 Number of Flows = 3

 Number of Memory Modules = 3

 Memory Contention Does Not Exist

 Algorithm to be used = DP

  ========================== System Configuration =======================
  Flow 1         -    Task1    Task2    Task3    Task4    Task5
  Execution Time -  5.000000  4.000000  3.000000  1.000000  3.000000

  Flow 2         -    Task10   Task12   Task13   Task14   Task15
  Execution Time -  5.000000  4.000000  3.000000  1.000000  3.000000

  Flow 3         -    Task6    Task7    Task8    Task9
  Execution Time -  5.000000  3.000000  4.000000  2.000000

  ====================== Dynamic Programming Allocation ===================
  Processor = 1 :  4  5  14  15  8  9
  Processor = 2 :  2  3  12  13  7
  Processor = 3 :  1  10  6

  Pipeline Throughput = 0.142857
 ==================================================================
```

Figure B.2: Sample - *Phoenix* Toolset Output

B.1.3 Output Format

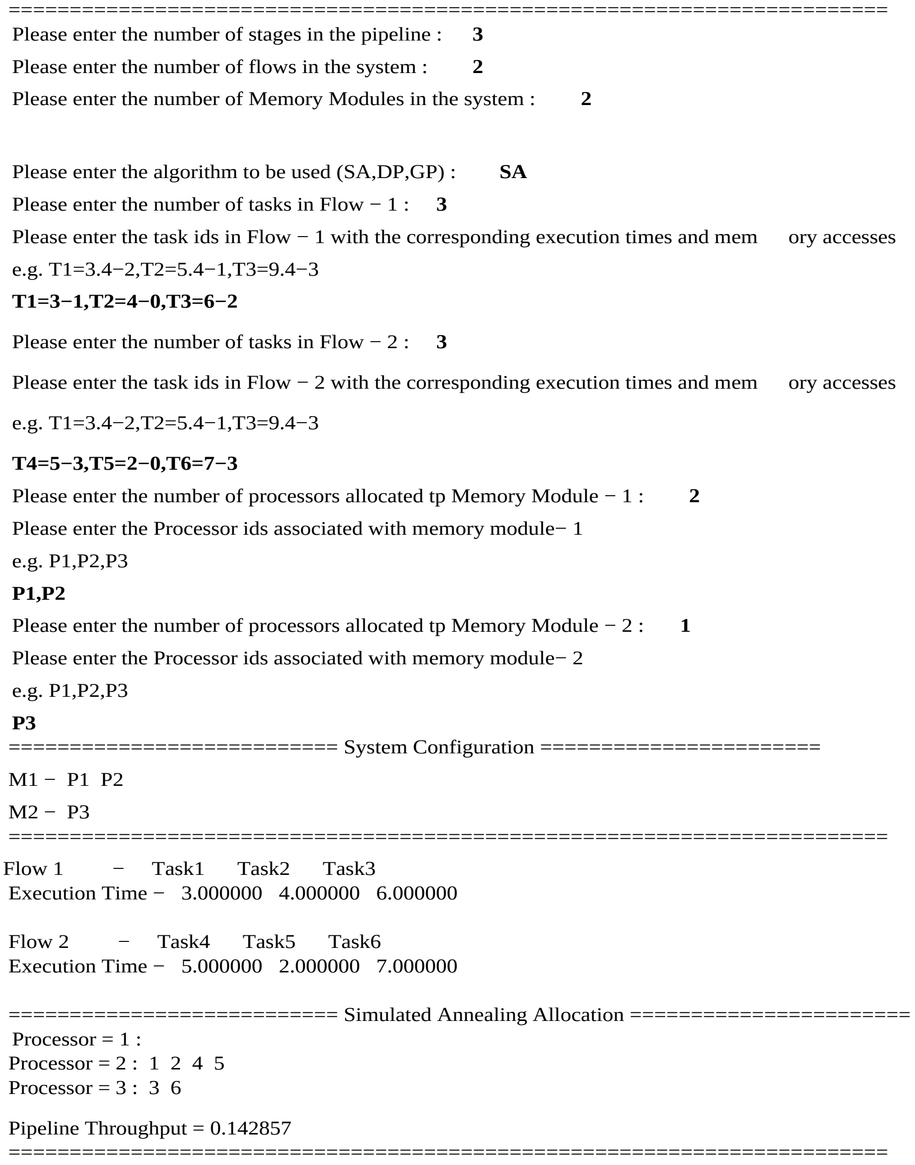
Given a set of input parameters for the system configuration, *Phoenix* determines and displays the optimal task assignments for the pipeline. Figure B.2 shows a sample output format. The displayed output (FigureB.2) shows the tasks allocated to each processor stage and also the throughput of the system for the given allocation.

B.2 Example

This section gives complete examples of execution of the tool using both the command line option and the the configuration file option. The first example being considered has a system configuration with 3 processor stages and two flow classifications. The applications for both the flows are pipelined into three tasks. There are two memory modules in the system; the first one has two processors allocated to it and the second one has one processor allocated to it. Since the memory modules are shared, there is memory contention. The algorithm to be used for task scheduling is SA. Figure B.3 shows all the steps of the tool execution when the described system configuration information is entered interactively through the command-line.

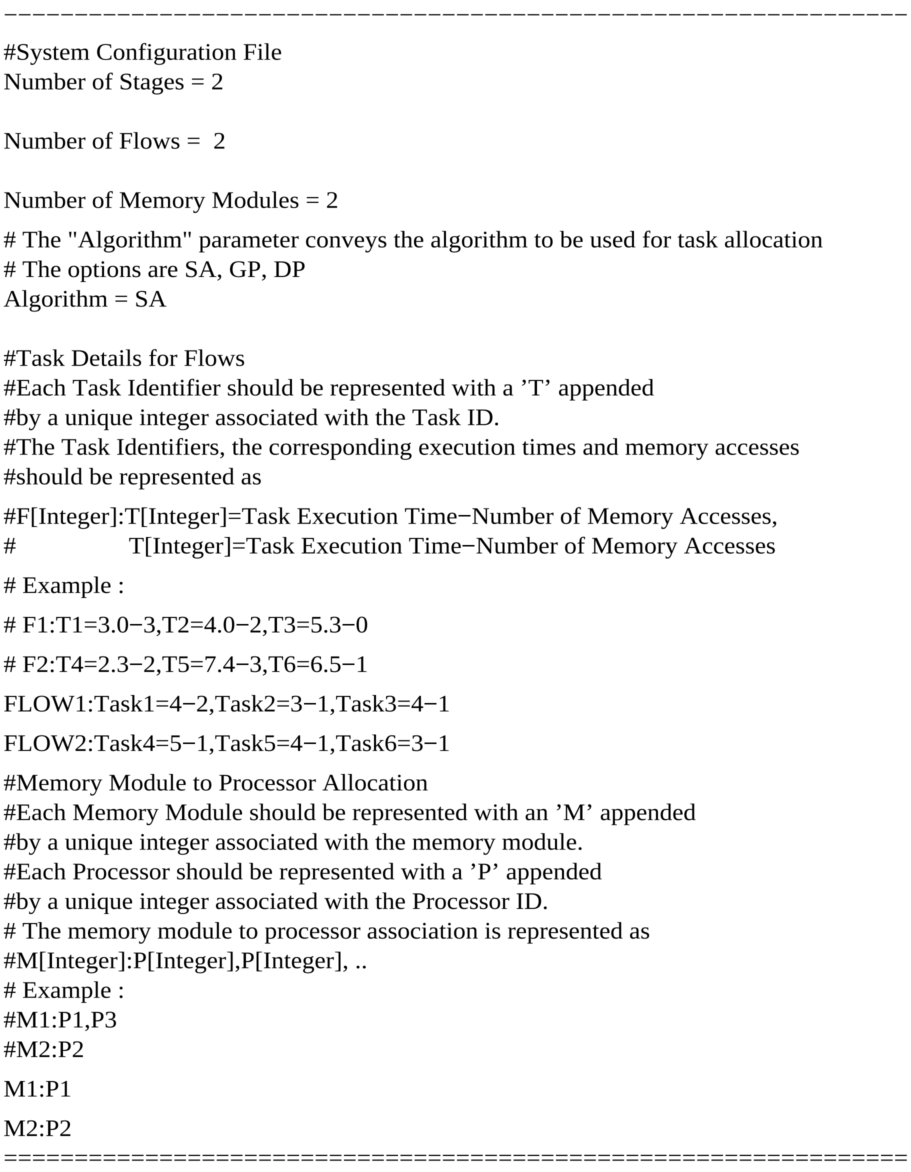
The next example being considered has the input parameters provided through the configuration file. The pipeline has two processor stages and two flows. The applications in both the flows are pipelined into three tasks. There is no memory contention. The algorithm to be used is SA. Figure B.4 shows the contents of the *config.txt* file that describes the pipeline configuration details.

Figure B.5 shows the allocation generated by *Phoenix* for the system configuration in Figure B.4.



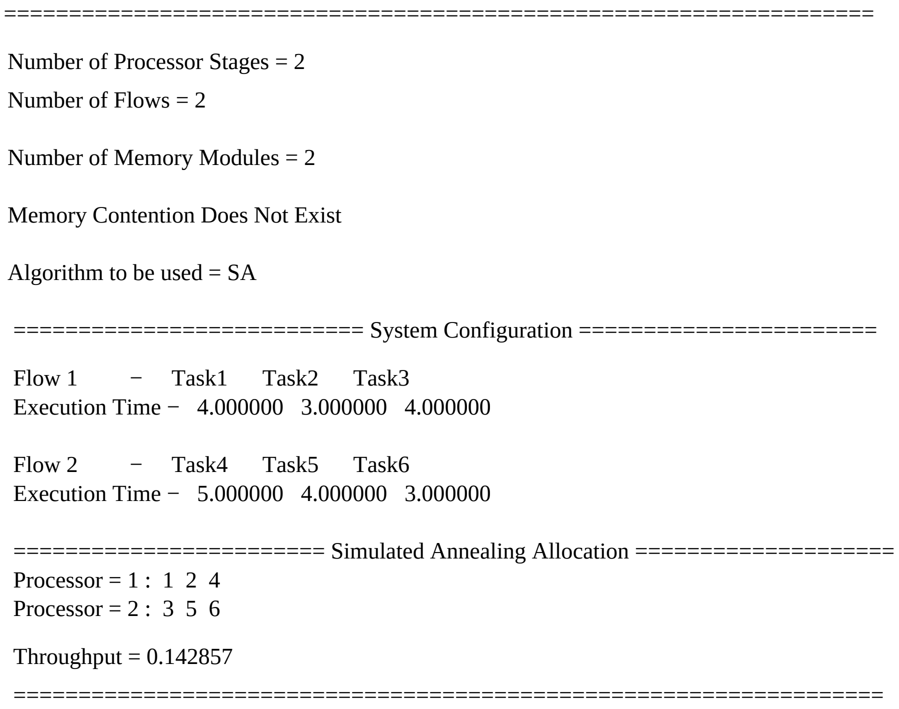
```
=============================================================================
 Please enter the number of stages in the pipeline :    3
 Please enter the number of flows in the system :       2
 Please enter the number of Memory Modules in the system :        2


 Please enter the algorithm to be used (SA,DP,GP) :       SA
 Please enter the number of tasks in Flow − 1 :    3
 Please enter the task ids in Flow − 1 with the corresponding execution times and mem      ory accesses
 e.g. T1=3.4−2,T2=5.4−1,T3=9.4−3
 T1=3−1,T2=4−0,T3=6−2
 Please enter the number of tasks in Flow − 2 :    3
 Please enter the task ids in Flow − 2 with the corresponding execution times and mem      ory accesses
 e.g. T1=3.4−2,T2=5.4−1,T3=9.4−3
 T4=5−3,T5=2−0,T6=7−3
 Please enter the number of processors allocated tp Memory Module − 1 :       2
 Please enter the Processor ids associated with memory module− 1
 e.g. P1,P2,P3
 P1,P2
 Please enter the number of processors allocated tp Memory Module − 2 :      1
 Please enter the Processor ids associated with memory module− 2
 e.g. P1,P2,P3
 P3
=========================== System Configuration =======================
M1 −  P1  P2
M2 −  P3
=============================================================================
Flow 1        −     Task1      Task2      Task3
Execution Time −   3.000000   4.000000   6.000000

Flow 2        −     Task4      Task5      Task6
Execution Time −   5.000000   2.000000   7.000000

=========================== Simulated Annealing Allocation ======================
 Processor = 1 :
Processor = 2 :  1  2  4  5
Processor = 3 :  3  6

Pipeline Throughput = 0.142857
=============================================================================
```

Figure B.3: Sample - *Phoenix* Interactive Execution



```
______________________________________________________________
#System Configuration File
Number of Stages = 2

Number of Flows =  2

Number of Memory Modules = 2
# The "Algorithm" parameter conveys the algorithm to be used for task allocation
# The options are SA, GP, DP
Algorithm = SA

#Task Details for Flows
#Each Task Identifier should be represented with a ’T’ appended
#by a unique integer associated with the Task ID.
#The Task Identifiers, the corresponding execution times and memory accesses
#should be represented as
#F[Integer]:T[Integer]=Task Execution Time−Number of Memory Accesses,
#                T[Integer]=Task Execution Time−Number of Memory Accesses
# Example :
# F1:T1=3.0−3,T2=4.0−2,T3=5.3−0
# F2:T4=2.3−2,T5=7.4−3,T6=6.5−1
FLOW1:Task1=4−2,Task2=3−1,Task3=4−1
FLOW2:Task4=5−1,Task5=4−1,Task6=3−1
#Memory Module to Processor Allocation
#Each Memory Module should be represented with an ’M’ appended
#by a unique integer associated with the memory module.
#Each Processor should be represented with a ’P’ appended
#by a unique integer associated with the Processor ID.
# The memory module to processor association is represented as
#M[Integer]:P[Integer],P[Integer], ..
# Example :
#M1:P1,P3
#M2:P2
M1:P1
M2:P2
==============================================================
```

Figure B.4: Example - config.txt



```
========================================================================

Number of Processor Stages = 2
Number of Flows = 2

Number of Memory Modules = 2

Memory Contention Does Not Exist

Algorithm to be used = SA

 ============================= System Configuration =========================

 Flow 1         −     Task1      Task2      Task3
 Execution Time −   4.000000   3.000000   4.000000

 Flow 2         −     Task4      Task5      Task6
 Execution Time −   5.000000   4.000000   3.000000

 ========================== Simulated Annealing Allocation ======================
 Processor = 1 :  1  2  4
 Processor = 2 :  3  5  6

 Throughput = 0.142857
 ========================================================================
```

Figure B.5: Example - *Phoenix* Output

Appendix C

Phoenix Toolset File Organisation

The toolset installation comes in the form of a tar-gzip file called *Phoenix.tar.gz*. On extracting the tar file, it forms a directory called *Phoenix-1.0*. In addition to the *Phoenix* toolset manual, this directory has all the source files including a Makefile to build an executable. The source files in the directory are :

- *common.h* : Contains the functions and global variables common to all the three algorithms.
- *dp.c* : Contains the functions associated with the dynamic programming algorithm.
- *stack_dp.h* : Contains the functions to maintain a stack for the dynamic programming algorithm.
- *sa.c* : Contains the functions associated with the simulated annealing algorithm.
- *ce.c* : Contains the functions associated with the complete enumeration algorithm.
- *greedypipe.c* : Contains the functions associated with *GreedyPipe* (the heuristic approach).

- *stack_gp.h* : Contains the functions to maintain a stack for the *GreedyPipe* algorithm.
- *phoenix.c* : Contains the main program that invokes the three algorithms.
- *Phoenix_manual.pdf* : *Phoenix* toolset manual

To install the tool, run the *make* command which builds the binary *phoenix*.

References

[1] AES Algorithm Rijndael Information. http://csrc.nist.gov/CryptoToolkit/aes/rijndael.

[2] CAIDA : The Cooperative Association for Internet Data Analyses . `http://www.caida.org`.

[3] AS1239 BGP Table Data. http://bgp.potaroo.net/1239/bgp-active.html, 2003.

[4] BGP Table Data. http://bgp.potaroo.net/, 2003.

[5] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modelling. In *IEEE Computer*, February 2002.

[6] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63, 1962.

[7] P. Bjørstad, F. Manne, T. Sørevik, and M. Vajteršic. Efficient matrix multiplication on SIMD computers. *SIAM J. Matrix Anal. Appl.*, 13(1):386–401, 1992.

[8] P. Chretienne, Jr. E. G. Coffman, J. K.Lenstra, and Z. Liu. *Scheduling Theory and its Applications*. John Wiley & Sons, Chichester, England, 1995.

[9] K. Claffy, G. Miller, and K. Thompson. The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone. Technical report, April 1998.

[10] F. Manne and B. Olstad. Efficient Partitioning of Sequences. volume 44, pages 1322–1326, 1995.

[11] M. A. Franklin and T. Wolf. A Network Processor Performance and Design Model with Benchmark Parameterization. In *Network Processor Design, Vol.1, by Crowley, P., Franklin, M., Hadimioglu, H. and Onufryk, P.* Morgan Kaufmann Publishers, Inc., San Francisco, CA., 2003.

[12] M.A. Franklin and T.Wolf. A Network Processor Performance and Design Model with Benchmark Parameterization. In *Proc. 1st Workshop on Network Processors, in conjunction with 8th Inter. Symp. on High Performance Computer Architecture (HPCA-8), Cambridge, MA.*, Feb 2002.

[13] M.A. Franklin and T.Wolf. Power Considerations in Network Processor Design. In *Proc. 2nd Workshop on Network Processors, in conjunction with 9th Inter. Symp. on High Performance Computer Architecture (HPCA-9), Cambridge, MA.*, Feb 2003.

[14] M.R. Garey and D.S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. Comput.*, 4:397–411, 1975.

[15] M. Gordon. A Stream Compiler for Communication-Exposed Architectures. MIT Tech. Memo TM-627, Cambridge, MA, March 2002.

[16] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Optimization and approximation in deterministic sequencing and scheduling : A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[17] Leung J. Y.-T. Han S., Hong D. On the asymptotic optimality of heuristic multiprocessor scheduling algorithms. Technical report, Univ. of Nebraska, Department of computer Science, 1992.

[18] P. Hansen and K.-W. Lih. Improved algorithms for partitioning problems in parallel, pipelined , and distributed computing. *IEEE Trans. Comput.*, 41:769–771, 1992.

[19] P. Helman. The principle of optimality in the design of efficient algorithms. *Journal of Mathematical Analysis and Applications*, 119:97–127, 1986.

[20] IBM Corp. IBM Power Network Processors, 2000.

[21] Intel Corp. Intel IXP 2800 Network Processor, 2000.

[22] A. S. Jain and S. Meeran. Deterministic Job-Shop Scheduling: Past, Present and Future. *European Jrnl. of Operational Research*, 113(2), 1999.

[23] V. Joshi and M. Franklin. SimplePipe: A Simulation Tool for Task Allocation and Design of Processor Pipelines with Application to Network Processors. Technical report, Washington Univ. in St. Louis, CSE Dept, (Pending).

[24] Eddie Kohler, Robert Morris, and Benjie Chen. Programming language optimizations for modular router configurations. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 251–263. ACM Press, 2002.

[25] M. Franklin and S. Datar. Pipeline Task Scheduling on Network Processors. In *Proc. of 3rd Workshop on Network Processors*, Feb. 2004.

[26] Fredrik Manne and Tor Sørevik. Optimal partitioning of sequences. *J. Alg.*, 19(2):235–249, 1995.

[27] John Marshall. Cisco Systems - Toaster2. In *Network Processor Design, Vol.1, by Crowley, P., Franklin, M., Hadimioglu, H. and Onufryk, P.* Morgan Kaufmann Publishers, Inc., San Francisco, CA., 2003.

[28] A. Moestedt and P. Sjodin. IP Address Lookup in Hardware for High-Speed Routing. In *Hot Interconnects*, August 1998.

[29] N. Metropolis and A.N. Rosenbluth and M.N. Rosenbluth and A.H. Teller and H. Teller. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[30] P. Chodowiec and P. Khuon and K. Gaj. Fast implementations of secret-key block ciphers using mixed inner- and outer-round pipelining. In *ACM SIGDA Inter. Symp. on Field Programmable Arrays (FPGA'01)*, Monterey, CA, Feb. 2001.

[31] Ali Pinar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *to appear in Journal of Parallel and Distributed Computing.*

[32] V. Rijmen and J. Daemen. The Block Cipher Rijndael. In *Proc. of the Third International Conference on Smart Card Research and Applications, CARDIS'98, LNCS 1820*, pages 277–284, 2000.

[33] S. Anily and A. Federgruen. Structured Partitioning Problems. *Operations Research*, 13:130–149, 1991.

[34] S. Kirkpatrick and C. Gelatt and P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–679, 1983.

[35] M. Schwehm and T. Walter. Mapping and Scheduling by Genetic Algorithms. In *Conf. on Algorithms and Hardware for Parallel Processing*, pages 832–841, 1994.

[36] S.H. Bokhari. Partitioning Problems in Parallel, Pipelined, and Distributed Computing. *IEEE Trans. Comput.*, 37:48–57, 1988.

[37] T.Wolf and M.A. Franklin. CommBench - A telecommunications benchmark for network processors. In *Proc. of IEEE Inter. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Austin, TX., Apr 2000.

[38] T.Wolf and M.A. Franklin. Design tradeoffs for embedded network processors. In *Proc. of Inter. Conf. on Architecture of Computing Systems (ARCS) (Lecture Notes in Computer Science)*, volume 2299, pages 149–164, Karlsruhe, Germany, Apr 2002.

[39] V. Sarkar and J. Hennessey. Compile-time Partitioning and Scheduling of Parallel Programs. In *In ACM SIGPLAN '86 Symp. on Compiler Construction*, pages 17–26, 1986.

[40] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high-speed prefix matching. *ACM Transactions on Computer Systems*, 19(4), November 2001.

[41] Wei-Je Huang and N. Saxena and E. J. McCluskey. A Reliable LZ Data Compressor on Reconfigurable Coprocessors. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 249–258, Napa Valley, California, April 2000.

[42] T. A. Welsh. A Technique for High-Performance Data Compression. *Computer*, 17(6), 1984.

[43] Tilman Wolf and Mark A. Franklin. Locality-Aware Predictive Scheduling of Network Processors. In *Proc. 2001 IEEE Inter. Symp. on Performance Analysis of Systems & Software*, Tucson, Arizona, Nov. 2001.

[44] D.F. Wong, H.W. Leong, and C.L. Liu. In *Simulated Annealing for VLSI Design.* Kluver Academic Publishers, Norwell, MA., 1998.

[45] T. Yang and A. Gerasoulis. A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors. In *Proc. of the 1992 Scalable High Performance Computing Conf.*, Williamsburg, VA, 1992.

[46] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978.

Vita

Seema Datar

Date of Birth June 11, 1973

Place of Birth Jabalpur, Madhya Pradesh, India

Degrees B.E. Electronics and Telecommunications, 1995, S.G.S. Institute of Tech. and Science, Indore, India.

Publications Mark A. Franklin, Seema Datar: Pipeline Task Scheduling on Network Processors *Proc. of 3rd Workshop on Network Processors*, Feb. 2004

August 2004