Performance Analysis of
Packet Classification Algorithms on
Network Processors

Deepa Srinivasan

A thesis presented to the faculty of the

OGI School of Science & Engineering

at Oregon Health & Science University

in partial fulfillment of the

requirements for the degree

Master of Science

in

Computer Science & Engineering

May, 2003

The thesis "Performance Analysis of Packet Classification Algorithms on Network Processors" has been examined and approved by the following Examination Committee:

Dr. Wu-chang Feng, Thesis Advisor
Assistant Professor, OGI School of Science and Engineering

Dr. Lois M.L. Delcambre
Professor, OGI School of Science and Engineering

Dr. Wu-chi Feng
Associate Professor, OGI School of Science and Engineering

For my parents

Acknowledgements

Throughout my degree program, I have had the good fortune to receive guidance and support from several people. I am especially grateful to:

- Dr. Wu-chang Feng, Assistant Professor, my thesis advisor for his guidance, patience and encouragement throughout this thesis and for partly funding my research work. Thank you Wu.

- Dr. Lois Delcambre, Professor and Dr. Wu-chi Feng, Associate Professor for examining my thesis and providing invaluable feedback. Thank you Lois. Thank you Wu-chi.

- Erik J. Johnson, Senior Network Software Engineer, Intel Corporation, for providing the microengine C code libraries I used in this thesis and for providing technical advice as I progressed in my research. Thank you Erik.

- George Inness, Andrew Kegel and Robert Smith, my managers at IBM Corporation for their encouragement and support for my graduate education and for approving most of the funding for my degree program. Thank you George. Thank you Andy. Thank you Robert.

Finally, my most special thanks go to my parents and sister Hemu for their constant encouragement through the years and to Balan for being there through the final phases of this thesis. Thank you Appa, Amma, Hemu and Kanna.

Table of Contents

Acknowledgements iv

Abstract x

1 Introduction 1

1.1 Motivation for study 4

1.2 Contributions 6

1.3 Related Work 6

1.4 Other considerations 7

1.4.1 Scope 7

1.4.2 Availability of real-world firewall rulesets 7

1.5 Completion criteria for research 7

2 Network Processor Overview 8

2.1 Hardware Architecture Overview 8

2.1.1 StrongARM core; microengines 8

2.1.2 Memory interfaces 10

2.1.3 The Federated Bus Interface 11

2.2 Software frameworks 11

3 Packet Classification Algorithms 13

3.1 Basic Search 13

3.2 Grid of tries 13

3.3 Cross-producting 14

3.4 PACARS 14

3.5 FIS-tree 14

3.6 Heuristics 15

3.6.1 Recursive Flow Classification 15

3.6.2 Hierarchical Intelligent Cuttings 15

3.6.3 Tuple Space Search 15
3.7 Bit Vector 16
3.8 Summary 16
4 Bit Vector algorithm 18
4.1 Working of the algorithm 18
4.2 Formal definition 20
4.3 Software implementation 22
4.3.1 Implementation verification in C 22
4.4 Mapping the algorithm to the IXP 25
4.4.1 Microengine allocation 25
4.4.2 Design Approach 1 26
4.4.3 Design Approach 2 26
4.4.4 Implementation/Design Details 27
4.4.4.1 Implementing BitVector1 27
4.4.4.2 Implementing BitVector2 29
4.4.4.3 Data Structure placement in memory 32
4.4.4.4 Verification of the implementations 32
4.5 Other considerations 34
4.5.1 Management application 34
4.5.2 Simulator 34
4.5.3 Scalability of the Bit Vector algorithm 35
5 Results and Analysis 36
5.1 Performance Results 37
5.1.1 Linear Search (LS) 38
5.1.2 Bit Vector Approach 1 (BV1) 39
5.1.3 Bit Vector Approach 2 (BV2) 40
5.1.4 Performance Comparison 41
5.2 Analysis 46
5.2.1 Linear Search vs. Bit Vector 46

5.2.2 BitVector1 vs. BitVector2 48

6 Conclusion 51

References 53

Biographical Sketch 55

List of Figures

1.1 Tradeoff between speed and programming ease of some design choices................2

1.2 Architecture block diagram of a parallel implementation (from the original paper [1]) of the Bit Vector algorithm...5

2.1 High level architecture of the Intel IXP1200...9

4.1 Projections of dimension values on the 1-axis..19

4.2 Projections of dimension values on the 2-axis..19

4.3 Projections of dimension values on the 3-axis..19

4.4 Bitmaps for each *j*-axis representing the rules matching each range on the axis...22

4.5 Pseudocode for classification phase implementation...................................24

4.6 Design Approach 1 of the Bit Vector algorithm 26

4.7 Design Approach 1 of the Bit Vector algorithm 27

4.9 Microengine responsibility and communication in BitVector1...29

4.10 Microengine responsibility and communication in BitVector2 31

5.1 Comparison of the receive and transmit rates of the algorithms 41

5.2 Comparison of memory access (%) of the algorithms...................................42

5.3 Comparison of memory access rates (Mbps) of the algorithms 42

5.4 Comparison of packets sent/receive ratio of the algorithms............................43

5.5 Comparison of microengines executing time...43

5.6 Comparison of microengines aborted time...44

5.7 Comparison of microengines idle time...44

5.8 Comparison of microengines execution in Mips 45

5.9 Distribution of individual microengine time for BitVector1...........................45

5.10 Distribution of individual microengine time for BitVector2..........................46

List of Tables

1.1: Examples of network processors available in the market today 3

2.1: Registers available to each microengine 9

2.2: Types of Memory Interfaces on IXP1200..11

3.1: Performance characteristics of packet classification algorithms
(adapted from [2]) 17

4.1: Example
ruleset..19

4.2: Memory allocation for BV1 and BV2...32

5.1: Performance Metrics Collected...37

5.2: Overall performance statistics (LS) 38

5.3: Performance statistics of individual microengines (LS) 39

5.4: Overall performance statistics (BV1) 39

5.5: Performance statistics of individual microengines (BV1) 40

5.6: Overall performance statistics (BV2) 40

5.7: Performance statistics of individual microengines (BV2) 41

Abstract

Performance Analysis of Packet Classification Algorithms on Network Processors

Deepa Srinivasan

M.S., OGI School of Science & Engineering
at Oregon Health & Science University

May 2003

Thesis Advisor: Dr. Wu-chang Feng

Packet classification is a fundamental function performed by networking devices such as routers, switches, and firewalls. Approaches for designing packet classification algorithms are typically based on the hardware they are run on. Programmable network processors are an emerging platform that aims to offer the flexibility of software while achieving performance comparable to custom hardware. A key architectural feature of typical network processors is that the hardware is highly optimized to provide parallel execution of code. This thesis studies how different design approaches of mapping a parallel-search packet classification algorithm onto a network processor may affect its performance/packet-processing speed. In particular, it analyzes two different mappings the Bit Vector algorithm on the Intel IXP1200 network processor. It presents performance results and analysis from implementing these two different designs; this analysis can be used as input while designing packet classification algorithms for network processors; or while designing similar algorithms for parallel hardware in general.

1 Introduction

Over the past two decades, computing environments have witnessed rapid transformations and progress in several areas resulting in what can be termed as a gigabit environment [12], in terms of CPU speed, memory sizes and disk sizes. Gigabit or high speed computer networking is an important component of this environment. As the processing speeds of components within a computer system are increasingly advancing to high speeds, it becomes essential that the networks connecting such systems also are capable of handling high speeds [13].

A wide range of applications, from file servers to fancy multimedia applications, benefit from high speed networks. Some of the specific applications are in scientific modeling & engineering, publications, medical transfer, Internet/intranets, data warehousing, network backups and desktop video conferencing. With widespread use of such applications, the demands on various components in the network - adapters, hubs, bridges, switches and routers - will need to be able to process at high speeds. This implies that each of the functions performed by these network components and the algorithms that are implemented for these functions have high operating speed requirements.

There are several choices available for use while designing such network devices. Some examples are General Purpose Processors (GPPs), embedded RISC processors, network processors, Field-Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs). For example, C. Partridge et al. [14] describe a router that can operate at a high speed of 50 Gbps (Gigabit per second), that was designed using custom hardware. In choosing from the available design options, we consider several tradeoffs including the speed of packet processing achievable and ease of programmability. This is illustrated in Figure 1.1.

At one end of the spectrum are ASICs that offer the fastest packet processing speed and are well-suited for the data plane. But they are not programmable, have a long time to market and are not reusable. At the other end, GPPs are programmable and have a mature software development environment. However, they are too slow to run the data plane effectively and are typically used to implement control plane functions. FPGAs are

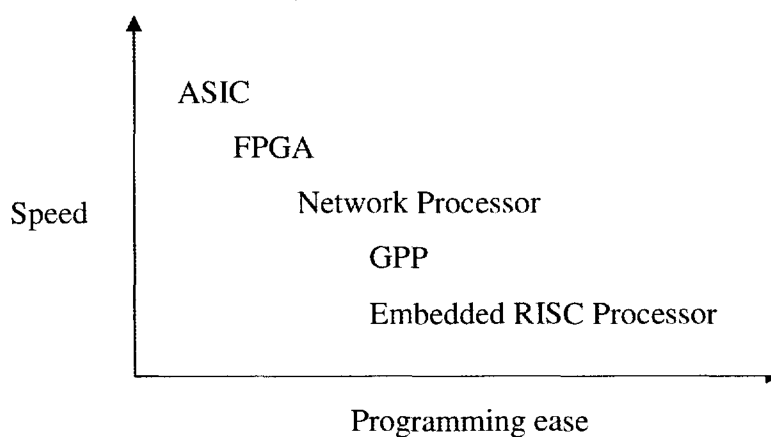

Figure 1.1: Tradeoff between speed and programming ease of some design choices

flexible re-programmable hardware which are less dense and slower than ASICs. However, they are good for providing fast custom functionality and well-suited for data plane functions. Embedded RISC processors offer the same level of programmability as GPPs; however they tend to be slower. They are typically designed specifically for network applications and are used for control plane functions.

Programmable network processors (NPs) are an emerging class of processors that aim to offer the flexibility of implementing networking services/applications in software as with GPPs/embedded RISC processors, while achieving high packet processing speed as with ASICs/FPGAs. They are flexible, reusable components with a lower cost than custom hardware. Several vendors offer families of network processors, each of which has a different hardware architecture that is suitable for use in different domains and have different operating speeds. Examples are the Intel IXP, IBM PowerNP and Motorola C-3

families of processors. Some of these are described in Table 1.1 [15, 16]. There are some architectural features that are common across different network processors. Multiple processing elements (PEs) are available with multiple hardware thread contexts per element. This enables thread context switches that have zero or minimal overhead. The instruction set of the processor is customized to suit network applications. The hardware architecture of network processors is described in detail in Chapter 2.

Table 1.1: Examples of network processors available in the market today

Network Processor	Description
Intel IXP1200	For applications operating at 155 Mbps to 622 Mbps
Intel IXP2400	For applications operating at 622 Mbps to 2.5 Gbps
Intel IXP422	For applications such as residential gateways, firewall appliances
IBM PowerNP NP4GS3	For applications operating at 4Gbps
IBM PowerNP NP2G	For applications operating at 2Gbps

Although NPs aim to offer programmability that is comparable to GPPs, there is one significant difference between developing software for NPs and GPPs – on the NP, the developer must have an in-depth understanding of the hardware architecture and of the various features available on a particular NP platform. Since there is no operating system available to perform dynamic resource management or load balancing on NPs, the binary image that is executed on a particular PE is pre-determined at compile/load time. Since an NP has multiple PEs, a single algorithm can be mapped in several different ways to each of the processing element and different mappings could have different performance/packet-processing speed. Hence, it is imperative that given an algorithm, the best possible mapping onto the NP platform is chosen for implementation.

An important class of algorithms used in networking is packet classification. Packet classification is a fundamental function performed by networking devices such as routers, bridges and switches and in networking applications such as firewalls and intrusion detection systems. It can be defined as the process of categorizing packets according to pre-defined *rules*. A *classifier* or *ruleset* typically refers to a collection of rules. Once a packet is received by a networking device or application, in order to classify it, the values in its header fields are looked up against those in the rules, until a match is found. Once the matching rule is found, the action specified by that rule is applied to the packet. Classification is typically performed based on the appropriate headers in a packet. We call each of the header fields a *dimension*. The complexity and/or the performance of the classification algorithms vary depending on the number of dimensions and number of rules used in the classification.

As the operating speeds of networks increase, the classification algorithms that are employed are required to process packets as close to wire-speed as possible, to prevent bottlenecks in networks. Network processors are ideally suited for use to implement classification at wire speed, since they aim to offer both programmability and wire-speed packet processing.

1.1 Motivation for study

In this thesis, we study the problem of mapping packet classification algorithms onto network processor hardware. The Intel IXA family of network processors is one of the predominant in the market today [17]. It is also provided to universities through the Intel IXA University program (http://www.ixaedu.com), along with the required development environment, for use in research projects. Hence, we choose the Intel IXP1200 (IXP) as our platform for study.

As is typical with NPs, the hardware architecture of the IXP is highly parallel (see Chapter 2 for details). Hence, an algorithm that is capable of performing the various stages of classifying a packet in parallel would be well suited for implementation on the

IXP. Let us consider the Bit Vector algorithm (see Chapter 4 for details). The original paper [1] describing the algorithm presents the diagram shown in Figure 1.2.

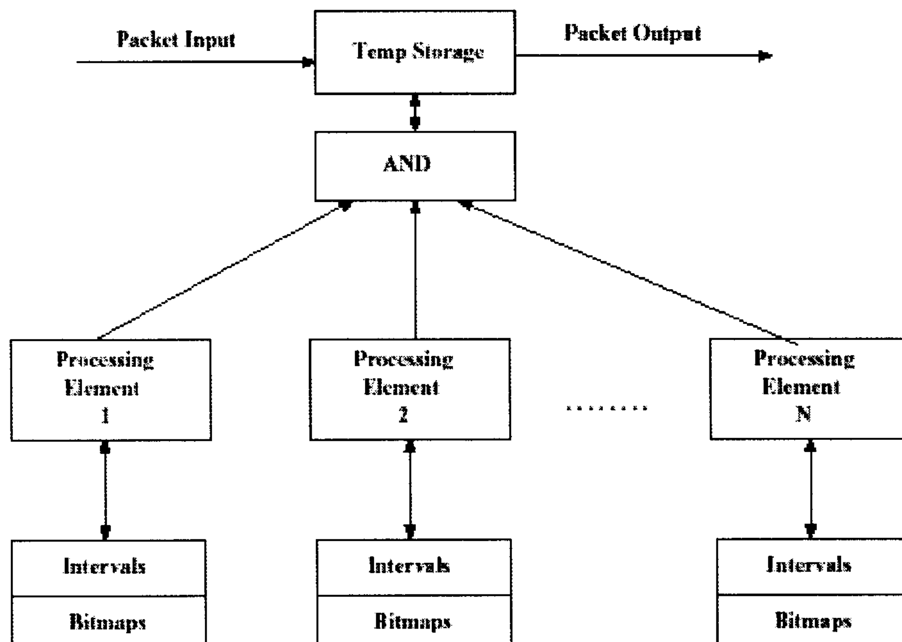

Figure 1.2 (from the original paper [1]): Architecture block diagram of a parallel implementation of the Bit Vector algorithm

It is indeed interesting to note the striking similarity of the architecture of an implementation of this algorithm and that of hardware architecture of a network processor (recall that a network processor has multiple PEs that can execute in parallel). Hence, it is natural that the Bit Vector algorithm could be mapped onto a NP such as the IXP easily. It may also seem that such an implementation will perform better than one where the algorithm is implemented sequentially on the IXP. However, our study shows that this is not true. There are several low-level implementation details that affect the performance of the algorithm on the IXP. This underscores the main idea in this thesis – that it is critical to choose the right design mapping of an algorithm onto the NP; while at

a higher level, it may seem that a particular mapping may perform better than another, the low-level implementation details and resource restrictions may prove otherwise.

1.2 Contributions

In this thesis, we propose different design mappings for implementing the Bit Vector algorithm on the IXP and present a comparison between two of the approaches. We present several critical performance metrics from the two implementations and analyze them in detail. Also, in order to have a baseline performance comparison, we implement a simple linear search algorithm and compare it to that of the Bit Vector algorithm, on the IXP. A principal observation we make is that a significant amount of a processing element's time on the IXP is wasted due to aborting its instruction execution pipeline, which happens typically due to branch instructions. This leads to the conclusion that algorithms with complex branch decisions will perform worse than those with simpler execution paths.

The results from this thesis can be used while designing packet classification or other class of algorithms specifically for programmable network processors with highly parallel hardware architectures.

1.3 Related Work

There are several packet classification algorithms that exist in current literature that propose various schemes for performing multi-dimensional classification. Chapter 3 presents a survey of these algorithms. However, they are not immediately applicable for implementation on network processors. There has been a recent study [5] that proposes designing efficient packet classification algorithms targeted for NP platforms, by exploiting various characteristics that are typically exhibited by classifiers. In this thesis, we argue that given any algorithm, it is important to choose the right mapping for implementation. This is critical since NPs aim to offer wire-speed packet processing. To our knowledge, there is no other previous work that studies this aspect of performance of packet classification algorithms on NPs at this level of detail.

1.4 Other considerations

1.4.1 Scope

The study of packet classification algorithms in general and their suitability for network processors is an extensive problem area. The core of this thesis is to evaluate two different design approaches of a single algorithm and present and compare performance metrics via implementation. We do not attempt to propose a new packet classification algorithm nor attempt to achieve the best possible performance of such algorithms on network processors.

1.4.2 Availability of real-world firewall rulesets

One of the most common applications of packet classification is in access control, as employed by firewalls. While designing a classification algorithm or evaluating its performance for real-world use, we need to use test rulesets that reflect characteristics. However, access control rules are meant to provide security to an enterprise's network and are not disclosed. Hence, we do not have access to such rulesets in the public domain. A recent study [5] of four such rulesets was done and the characteristics were published. However, in this thesis, we analyze the performance of an algorithm independent of the input ruleset – that is, given a fixed ruleset, we study the performance of different design mappings. One of the future steps in this work might be to generate a synthetic ruleset database that follows the characteristics typically exhibited by real-world rulesets [5] and study the performance. In this thesis, we study the performance only against a small sample ruleset.

1.5 Completion criteria for research

The completion criteria of this thesis is in achieving correct implementations of the two approaches of the Bit Vector algorithm and the linear search algorithms on the IXP, running simulations of packet streams through these implementations, recording performance statistics of these implementations and analyzing the same.

2 Network Processor Overview

This chapter presents the hardware architecture and software development environment of the Intel IXP1200 (chosen for study in this thesis).

2.1 Hardware Architecture Overview

The IXP1200 is an integrated network processor, comprised of a single StrongARM processor, six microengines, standard memory interfaces and high-speed bus interfaces. Complete and detailed description of the hardware architecture is available from other sources [3, 4]. Figure 2.1 shows the high-level architecture of the IXP1200 [18].

2.1.1 StrongARM core; microengines

The StrongARM core is an Advanced Reduced Instruction Set Computer (RISC) Machines (ARM) general-purpose processor. The StrongARM core offers a tradeoff between computational capability and power consumption. The six microengines are RISC processors optimized for fast-path packet processing and have an instruction set tuned for processing network data. Each instruction on the microengines takes up one long word (32 bits) of control store space; each microengine has an independent 8 KB instruction store, enough for 2048 instructions.

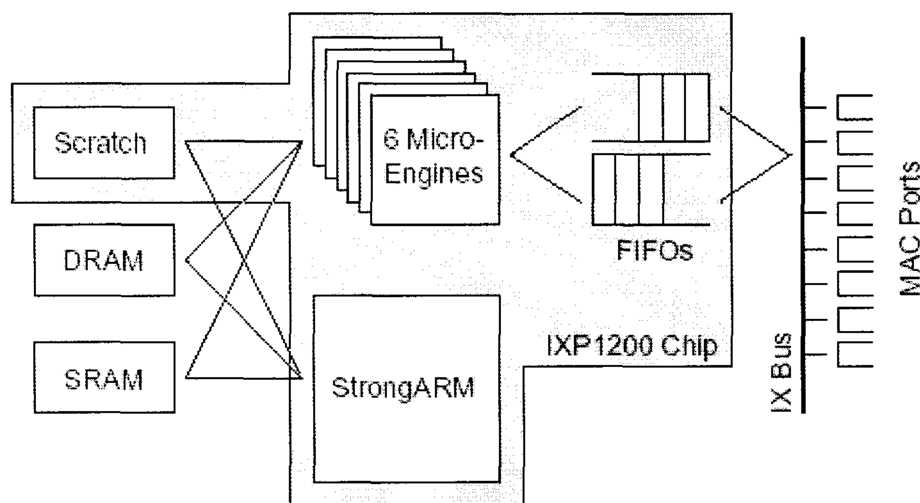

Figure 2.1: High level architecture of the Intel IXP1200

On the microengines, instructions are executed in a five-stage pipeline. Each microengine has four hardware-assisted threads of execution. All threads in a particular microengine execute code from the same instruction store on that microengine. An equal portion of the total registers available to a microengine is allocated to each hardware thread context; hence, registers of each microengine need not be flushed to memory on a thread context switch. The thread arbiter on the microengine is non-preemptive and swaps between threads in a round-robin order. Communication between threads in a microengine is done using registers; communication across microengines is done using the shared SRAM and SDRAM memory.

A key feature of the IXP1200 is asynchronous memory access – for example, a microengine thread can issue a memory request and then continue processing; the completion of the request could then be asynchronously reported back to the microengine thread.

There are different kinds of registers on the IXP1200. Table 2.1 lists the types of registers and their purposes.

Table 2.1: Registers available to each microengine

Type of Register	Number of registers per microengine	Description
General Purpose	128	32-bit registers; allocated into two banks of 64 each.
SRAM transfer	64	Used to transfer data to and from the SRAM unit, the IX Bus interface and PCI interface.
SDRAM transfer	64	Used to transfer data to and from the SDRAM unit.

The IXP1200 development platform consists of the network processor card which is a PCI card that fits into the host system. This section describes the architecture of the network processor card. The main components of the card are a StrongARM processor known as the core processor, six microengines, SDRAM memory, SRAM memory unit and the IX bus interface unit. Figure x presents this architecture.

2.1.2 Memory interfaces

The IXP contains three different memory interfaces: scratchpad, SRAM and SDRAM. Table 2.2 summarizes the details of each of these. In addition, each interface provides some additional features. The scratchpad unit provides atomic increment and bit test-and-set/clear primitives. The SRAM unit provides CAM locks, atomic bit test-and-set/clear and push-pop queues. The SDRAM unit provides a direct path to and from the Federated Bus Interface (FBI) which allows data to be moved between the two without first going through one of the processors.

Table 2.2: Types of Memory Interfaces on IXP1200

Memory Interface	Minimum Addressable Units (Bytes)	Addressable Size	Actual Size	Approx. unloaded latency (clocks)
Scratchpad	4	1 KB	4 KB	12-14
SRAM	4	2 MB	8 MB	16-20
SDRAM	8	32 MB	256 MB	33-40

2.1.3 The Federated Bus Interface

The main purpose of the FBI is to provide an interface for receiving and transmitting packets. The FBI holds the on-chip scratchpad memory, contains a set of control and status registers, a hash generation unit and the interface to the IX bus.

2.2 Software frameworks

Intel provides the IXP Software Development Kit (SDK) for use by software developers to implement network services on the IXP. There are two basic programming choices in the SDK – programming in microcode/assembly language or programming in microC. Microengine C or microC [19] is a C-like language that includes features for programming on the IXP. For example, the microC compiler implements intrinsic functions that support features of the IXP1200 that are not accessible using standard C; intrinsics are built into the compiler (they are not library functions). An example of an intrinsic is the *sdram_r_fifo_read_ind* that reads (packet) data from a receive FIFO and places it in SDRAM memory

Intel also defines the MicroACE framework that should be used by developers while writing code targeted to run on the IXP hardware. Using the MicroACE framework, code is organized into ACEs or Active Computing Elements. Each ACE is composed of a microblock that runs on microengines and a core block that runs on the StrongARM core. Communication between these blocks is defined by the ACE framework.

The code that we use in this thesis implements microblocks using microC. This SDK is provided by Intel on a limited basis to limited customers and does not have documentation or support. However, the code and framework is very similar to microcode microACE and hence can be worked with, after the initial learning curve.

3 Packet Classification Algorithms

This chapter presents a survey of several packet classification algorithms that exist in current literature. These algorithms can be classified as basic-search, geometric, trie-based, hash-based, parallel-search and heuristic [2], [3]. Each of the algorithms has different time and space complexities. The Bit Vector algorithm, chosen for study in this thesis, is described in detail in Chapter 4. The following sections describe each of the other algorithms briefly. We use the following terminology in this thesis:

N – represents the number of rules in a classifier.

W – represents the width of each field in a packet header.

d – represents the number of dimensions in each packet that is used for classification.

3.1 Basic Search

A linear search is an example of a basic search algorithm. Rules can be represented by a linked list that can be ordered by decreasing priority. To find a matching rule for a packet, we simply do a linear search of the linked list. This algorithm has $O(Nd)$ space complexity and $O(Nd)$ time complexity.

3.2 Grid of tries

Packet classification at network Layer-4 (i.e. the algorithms can be used for classifying based on headers available at Layer-4 of the OSI architecture) can be done using two different approaches [6]. One type of data structure that can be used for packet classification is set-pruning trees. For example, for two-dimensional classification (say, source and destination IP addresses), a set pruning tree is constructed as follows: Build a trie containing destination prefixes; each valid prefix in a destination trie points to a trie containing some source prefixes. While searching, we first match the destination of the packet header in the destination trie; we then traverse the associated source trie to find the longest source match. However, this scheme has a memory blowup problem – the memory requirement for this scheme (for d dimensions) is $O(N^d dW)$, since each rule may have to be stored dW times.

In the grid of tries algorithm, the worst-case lookup time can be improved by storing pre-computed switch pointers in the trie – the switch pointer can be used to speed up search in a later source trie based on a search in an earlier source trie. This has a time complexity of $O(W)$ and memory complexity of $O(NW)$. However, switch pointers make incremental updates to the grid of tries difficult.

3.3 Cross-producting

A second approach for Layer-4 classification is *cross-producting* [6]. The main idea behind cross-producting is to: slice the filter database into columns, with the ith column representing all the distinct filters in field i; when a packet P is to be classified, the best matching prefix for each of the its fields is determined separately and the individual lookup results are combined. The basic cross-producting algorithm is to build a table of all possible cross-products and precompute the least cost filter matching each cross-product. This gives a time complexity of $O(dW)$ and space complexity of $O(N^d)$.

3.4 PACARS

Another class of algorithms is **Pa**cket **C**lassification **A**lgorithms using **R**ecursive **S**pace-decompositions (PACARS) [7]. A specific instance of this is the Area-based Quad Tree (AQT). For 2-dimensional classification, this algorithm has a space complexity of $O(N)$ and search time complexity of $O(\alpha W)$, where α is a tunable parameter. The worst-case update time is $O(\alpha \sqrt[\alpha]{N})$.

3.5 FIS-tree

Packet classification can be done using a data structure called the Fat Inverted Segment tree or FIS-tree [8]. Given a set S of possibly overlapping line segments, a segment tree is a balanced binary search tree containing the end points of the line segments in S. It can be used for operations such as determining the highest priority line segment containing a given point [2]. An FIS-tree is a modified segment tree: (a) the segment tree is compressed to decrease its depths and occupies a given number of levels l; (b) up-

pointers from child to parent are used. The storage complexity using FIS-trees is $O(l \times N^{1+1/l})$; the search time complexity is $O((l+1)W)$ [2]. The tradeoff between storage space requirement and search time can be achieved by changing the value of l.

3.6 Heuristics

3.6.1 Recursive Flow Classification

Heuristic algorithms exploit the structure and redundancy that is observed in packet classification rules. The classification problem can be stated as mapping S bits in the packet header to a T bit action identifier, where $T = log\ N$ and $T \ll S$ [2]. Recursive Flow Classification [9] attempts to perform the mapping over several phases – in each phase, one set of values are mapped to a smaller set.

3.6.2 Hierarchical Intelligent Cuttings

Another heuristic algorithm that exploits the structure of the classifier is Hierarchical Intelligent Cuttings or HiCuts [10]. Each node in the HiCuts tree stores a small number of rules that can be searched sequentially to find the best match. The characteristics of this tree are chosen while preprocessing the classifier, based on its structure.

Both of the above algorithms have a storage complexity of $O(N^d)$ and search time complexity of $O(d)$.

3.6.3 Tuple Space Search

The Tuple Space Search algorithm [11] is motivated by the observation that while filter databases contain many different prefixes or ranges, the number of distinct prefix lengths tends to be small. This implies that the number of distinct combinations of prefix lengths also tends to be small. A *tuple space* is defined as the set of tuples defined for each combination of field length. The set of rules mapped to the same tuple can be stored in a hash table. Given a packet, all the tuples in the tuple set can be linearly probed to determine the least cost filter matching that packet. The search cost is proportional to m,

the number of distinct tuples; storage complexity is $O(N)$. Optimizations such as tuple pruning and precomputation can also be applied to the basic Tuple Space Search algorithm.

3.7 Bit Vector

We present it here for sake of completeness in the survey of packet classification algorithms. The basic idea behind the Bit Vector algorithm is to preprocess the ruleset to produce *P-sets* and *bit vectors* (see Chapter 5). The *P-sets* are used for lookup in the first step of classification. The results from this lookup are then used to perform an intersection of the corresponding *bit vectors* to determine the matching rule. The storage complexity for this algorithm is $O(dN^2)$ and the time complexity is $O(dW + N/mw)$, where *mw* is the memory width, or the number of bits that can be accessed in one memory read. Optimizations can be applied to the algorithm by using incremental reads to reduce the space requirements and by using a specialized scheme for 2-dimensional classification. The Aggregated Bit Vector scheme [11] makes the basic Bit Vector algorithm more scalable by introducing aggregation of data structures and rearrangement of rules.

3.8 Summary

This section summarizes the performance characteristics (presented in Table 3.1) of the various packet classification algorithms surveyed in this chapter. (Table 3.1 is adapted from [2]).

Table 3.1: Performance characteristics of packet classification algorithms (adapted from [2])

Algorithm	**Worst-case time complexity**	**Worst-case storage complexity**
Linear Search	Nd	Nd
Set-pruning Tries	dW	$N^d dW$
Grid of tries	W^{d-1}	NdW
Cross producting	dW	N^d
Area-based quad tree	αW	N
Fat Inverted Segment tree	$(l + 1)W$	$l \times N^{1+1/l}$
Recursive Flow Classification	d	N^d
Hierarchical Intelligent Cuttings	d	N^d
Tuple Space Search	m	N
Bit Vector	dW + N/memory-width	dN^2

Legend for Table 3.1

N: number of rules in the classifier

d: number of dimensions for classification

W: maximum number of bits in each field for classification

α: tunable parameter used in Area-based quad tree algorithm

l : number of levels occupied by a FIS-tree

4 Bit Vector algorithm

In this chapter, we describe the Bit Vector algorithm in detail, propose different design mappings of this algorithm onto the IXP and describe the software implementation.

4.1 *Working of the algorithm*

In any packet classification algorithm, there are typically two distinct phases. In the preprocessing phase, the ruleset is processed and data structures are built in memory, that are used in the second phase. The second phase is the actual classification phase, where these data structures are looked up to determine the matching rule. We now explain both phases of the Bit Vector algorithm using an example (the formal definition of the algorithm is given in Section 4.2). Consider the ruleset shown in Table 4.1. We assume it is ordered in decreasing priority.

Table 4.1: Example ruleset

Rule	Field 1	Field 2	Field 3	Action
r_1	(10, 11)	(2, 4)	(8, 11)	Allow
r_2	(4, 6)	(8, 11)	(1, 4)	Allow
r_3	(9, 11)	(5, 7)	(12, 14)	Deny
r_4	(6, 8)	(1, 3)	(5, 9)	Allow

Preprocessing phase

Assume that j represents a dimension, with $1 <= j <= d$ (in the example ruleset, $d = 3$). First, we take the jth dimension from all the rules and project them on the j-axis. For example, we take the values of dimensions in the first field of all rules and project them on the 1-axis, as shown in Figure 4.1. Note that rules may be overlapping in dimensions. We get distinct regions on this axis, and we note the rules for each region. Similarly, we do this for the other two dimensions (shown in Figures 4.2 and 4.3).

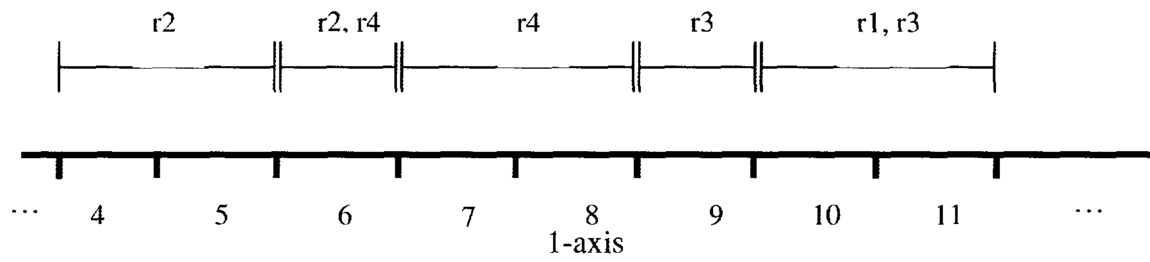

Figure 4.1: Projections of dimension values on the 1-axis

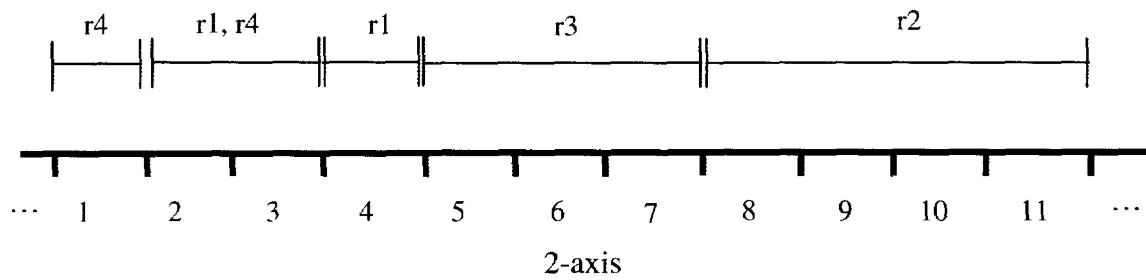

Figure 4.2: Projections of dimension values on the 2-axis

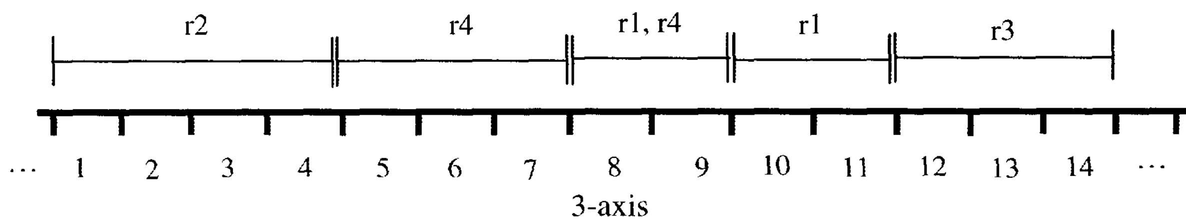

Figure 4.3: Projections of dimension values on the 3-axis

Classification phase

Consider that we have a packet whose fields are represented by the set { E_1, E_2 ... E_d }. For example, let us take a packet represented by P = { 6, 10, 2 } and classify it against the ruleset shown above. The classification phase for the Bit Vector algorithm consists of two stages.

- In the first stage, we take each dimension's value and determine the range which it matches on the corresponding axis. For example, we take the value 6 in the first dimension from packet P. We determine which range it matches on the 1-axis (Figure 5.1). In this case, it matches the second range, which corresponds to rules r_2 and r_4. Similarly, we do this for the remaining ranges. For packet P, the second dimension value (10) matches the range on the 2-axis corresponding to r_2; the third dimension value (2) matches the range on the 3-axis corresponding to r_2. This range matching operation for each dimension can be performed in parallel; hence Bit Vector is a parallel search algorithm.
- In the second stage, we take the results of the range matching operations and determine the common rule for all the dimensions. In the example above, r_2 is the common rule and hence it is the rule that matches the packet. The action specified by this rule is then applied to the packet. If more than one rule is common, then the rule with the highest priority is chosen.

While implementing the algorithm, the rule numbers matching each range can be represented using an array of bits, hence the algorithm is called "Bit Vector".

4.2 Formal definition

We now give the formal definition of the algorithm. Let there be a set of N packet filtering rules in *d* dimensions (the condition $1 <= j <= d$ holds, wherever *j* is specified in this definition). Each rule r_m is represented as the set $r_m = \{e_{1,m}, e_{2,m} \ldots e_{d,m}\}$, where each $e_{j,m}$ is a range of values (a, b) where a <= b, for the jth dimension. We assume that the rules are sorted based on their priorities.

In the preprocessing step of the algorithm, we perform the following operations to build the data structures that are used during classification.

(The condition 1 <= *j* <= *d* holds, wherever *j* is specified in the steps below)

1. For each dimension *j*, project all intervals $e_{j,i}$ (1 <= *i* <= N) on the j-axis, by extracting the jth element of every filter rule. There are a maximum of 2N + 1 non-overlapping intervals that are created on each axis. Let P_j denote the *d* sets of such intervals. We use the general term *P-sets* to describe the collection of all such sets P_j.
2. For each interval *k* in Pj, create sets of rules $R_{k,j}$ (1 <= *k* <= 2N + 1) such that a rule r_m belongs to the set $R_{k,j}$, if and only if, the corresponding interval *k* overlaps in the jth dimension with $e_{j,m}$.

The classification phase of the algorithm is as follows. Assume that a packet with fields $E_1, E_2 \ldots E_d$ needs to be classified.

1. For each dimension *j*, find the interval k_j in set P_j to which E_j belongs. This can be done using a binary search or any other efficient search algorithm.
2. Create the intersection of all sets $R_{kj,j}$, where k_j belongs to {1, 2, … 2N + 1}. This is done by taking the conjunction of the corresponding bit vectors in the bit arrays associated with each dimension and then determining the highest priority entry in the resultant bit vector.
3. The rule corresponding to the highest priority entry is the rule that matches the packet.

As mentioned in Section 4.1, the execution of this algorithm can be accelerated by taking advantage of bit-level parallelism. Each set $R_{k,j}$ can be represented by a bitmap N-bits long which acts as an indicator function for that set. Let B_j[k,m] be a (2N + 1) x N array of bits for each dimension *j*. The set $R_{k,j}$ can be stored as a bit vector in row *k* of the bit array – bit Bj[k,m] is set if and only if the rule r_m belongs to the set $R_{k,j}$. The intersection operation (Step 2) of the classification phase is then reduced to a logical AND among the

bitmaps that are retrieved after the search in Step 1. The bitmaps are organized in memory into words of width *w*, the unit of memory access being the word. The bitmaps for the example described in Section 4.1 is shown in Figure 4.4.

B_1 (1-axis)

	r_1	r_2	r_3	r_4
1	0	1	0	0
2	0	1	0	1
3	0	0	0	1
4	0	0	1	0
5	1	0	1	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0

B_2 (2-axis)

	r_1	r_2	r_3	r_4
1	0	0	0	1
2	1	0	0	1
3	1	0	0	0
4	0	0	1	0
5	0	1	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0

B_3 (3-axis)

	r_1	r_2	r_3	r_4
1	0	1	0	0
2	0	0	0	1
3	1	0	0	1
4	1	0	0	0
5	0	0	1	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0

Figure 4.4: Bitmaps for each *j*-axis representing the rules matching each range on the axis

4.3 Software implementation

The original implementation [1] of the Bit Vector algorithm was done in hardware. However, in order to map it onto the IXP, we need to implement it in software.

4.3.1 Implementation verification in C

We first implement the algorithm in C, both to understand and verify a software implementation of the Bit Vector algorithm. We assume the preprocessing phase of the algorithm to be performed manually and the data structures are initialized at program start. This is a valid assumption since this thesis focuses only on the microengine performance of the IXP which is responsible for the classification phase. The

preprocessing phase would typically be performed by a management application running on the StrongARM core processor. The pseudocode for the implementation of the classification phase in C is listed in Figure 4.5. The data structure initializations are simply to insert values into *P-sets* which are implemented as binary tries and into the bit vectors which are implemented as two-dimensional byte arrays.

This implementation was verified by passing different packet header values to the code and determining that the packet was matched against the correct rule.

```
//Get the index i for each P-set. (DIMS is the number of dimensions)
for(i = 0; i < DIMS; i++)
{
        range[i] = trieFind(PSet[i], packet_header[i]);
        if(range[i] == NOT_FOUND) goto exit;
}
//Get the corresponding bit vector
//Perform a logical AND on all bit vectors to determine the rule
result = bitVector[0][range[0] - 1];
for(i = 1; i < DIMS; i++)
{
        result &= bitVector[i][range[i] - 1];
}
//Determine which bit is set
result = (result << 4);
mask = 0x80;
for(i = 0; i < RULES; i++)
{
        if((result & mask) == mask)
        {
                matchingRule = i;
                break;
        }
        result <<= 1;
}
matchingRule++;
```

Figure 4.5: Pseudocode for classification phase implementation

This implementation was then ported to microC which is the programming language used for the IXP. Two different design approaches were implemented on the IXP, however the basic algorithm for classification remains the same. This is described in the next section.

4.4 Mapping the algorithm to the IXP

While programming the IXP, the developer has to be familiar with the underlying hardware architecture of the IXP so as to be able to extract optimal performance from it. The same algorithm could be mapped in several different ways onto the IXP so as to take advantage of the various hardware units available and the parallel nature of the hardware, specifically the six microengines which can execute instructions in parallel. The main motivation for study in this thesis is to determine how different mappings of the Bit Vector algorithm onto the IXP affect its performance characteristics and to study in detail the behavior of each of the mappings.

4.4.1 Microengine allocation

The code that is executed by each microengine is pre-determined at compile and link time. Hence, we need to partition the responsibilities prior to implementing and running the code on the IXP. The two standard functions that will be required are receiving and transmitting packets. In all our implementations, we allocate microengine 0 and microengine 5 for receiving and transmitting packets respectively. That gives us four microengines to use in the classification phase; the implementation of this phase can be done in different ways. The following sections list two such mappings. First, we introduce the terminology used for the different microengines: microengines that perform the classification function are called *cls* microengines; those that perform the receive and transmit functions are called *rcv* and *xmit* microengines respectively.

4.4.2 Design Approach 1

In this approach, all the classification steps for a single packet are performed by a single hardware thread in one microengine. This is illustrated in Figure 4.6. We call this design approach BitVector1 or BV1.

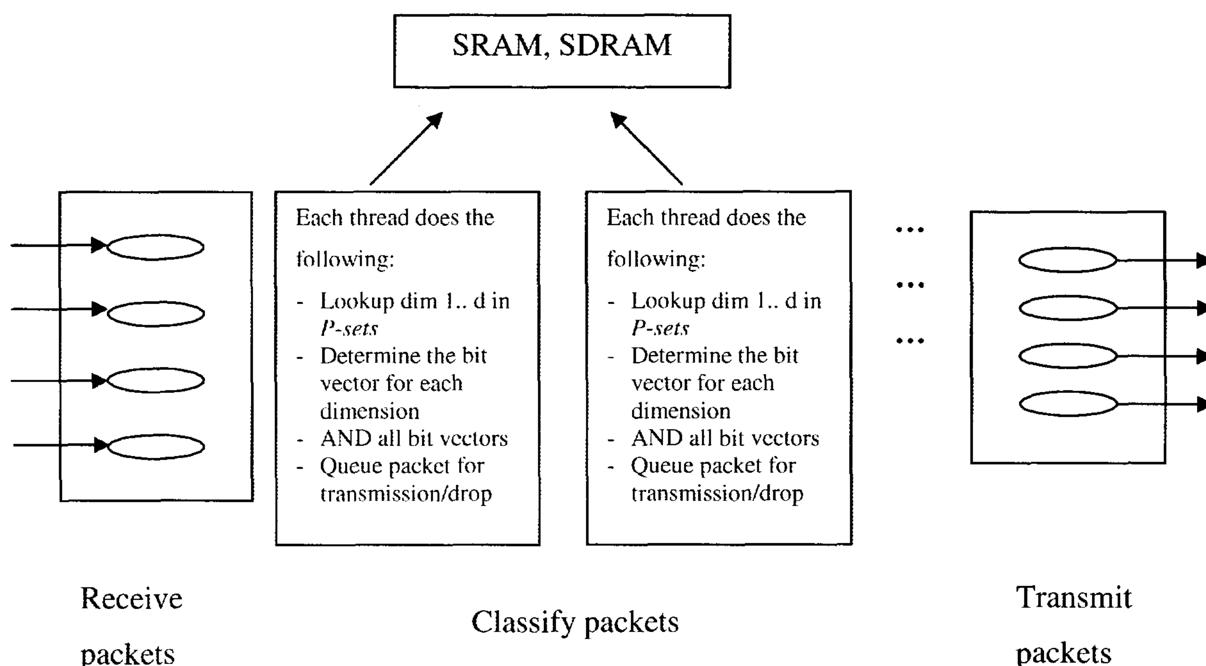

Figure 4.6: Design Approach 1 of the Bit Vector algorithm

4.4.3 Design Approach 2

In this approach, the first step (lookup in the *P-set*) of classification for a packet is done in parallel by multiple microengines. Each microengine performs the lookup for one particular dimension. We call such a microengine a *cls* (classification) microengine. For example, microengine 1 determines the range in the *P-set* for dimension 1, microengine 2 determines the same for dimension 2 and so on. At any given time, a single *cls* engine can perform a 1-dimension *P-set* lookup for 4 packets. The results of these lookups are then sent to a different microengine (say microengine 4) which then retrieves the

appropriate bit vectors and performs the logical AND operation. It then determines the matching rule and performs the appropriate action (transmit/drop). This design approach is illustrated in Figure 4.7. We call this design approach BitVector2 or BV2.

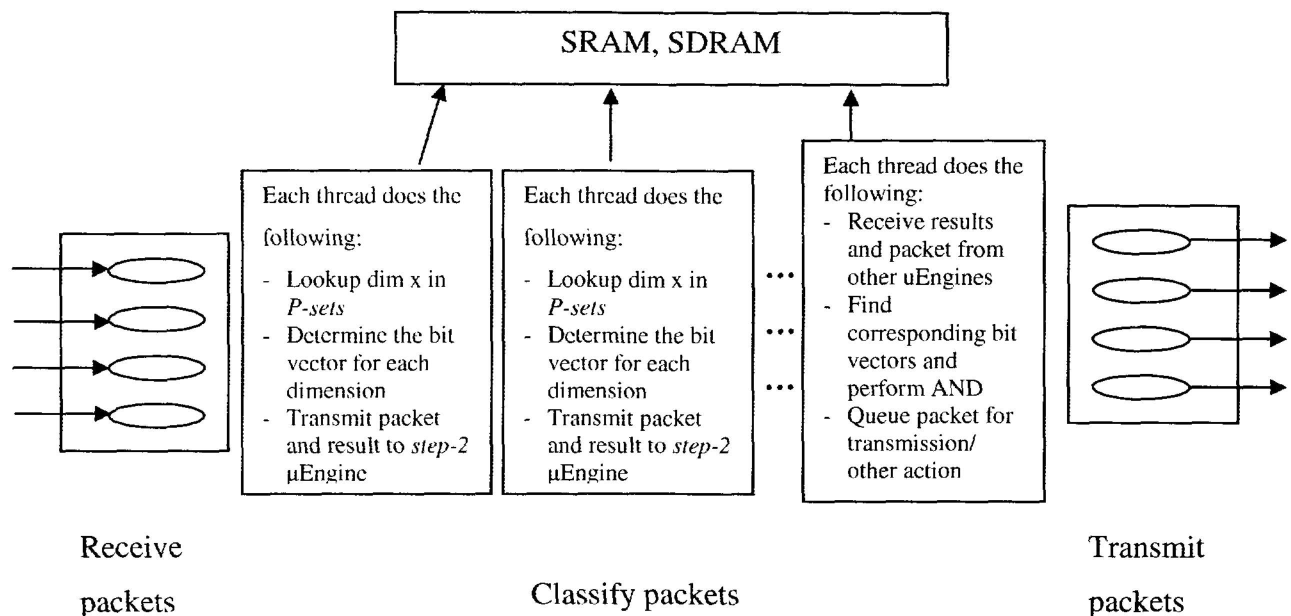

Figure 4.7: Design Approach 2 of the Bit Vector algorithm

4.4.4 Implementation/Design Details

This section describes the implementation details such as buffering schemes used, data structure placement in memory, intra and inter microengine communication, data structure initialization, number of ports available for receive/transmit, for BitVector1 and BitVector2.

4.4.4.1 Implementing BitVector1

Microengines 0 and 5 perform the functions of receive and transmit packets respectively. The code for this follows standard implementation – we reuse the code from the microC microACE sample for this, with small modifications as follows. Each of the four hardware threads on microengine 0 receives packets from a single port and queues them

for use by microengines 1 through 4. The queues used for this are circular and are placed in SRAM. Since there are four *cls* engines and all four perform the full classification for a packet, we create a queue for each of the *cls* hardware threads. Each of the four *rcv* threads rotates through the four queue numbers sequentially.

There are 128 entries in each of the queues with each entry occupying 2 words or 8 bytes of SRAM memory. The threads in the *cls* microengines wait for a new entry in their respective queue. Once an entry is available, it reads the appropriate packet headers, performs the classification and queues it for transmission by one of the *xmit* threads. Similar to the *rcv* threads, there are 4 *xmit* threads that service the 16 *cls* threads. Hence, each *xmit* thread rotates through transmitting packets from the 4 queues that are allocated to it. The detailed division of responsibility and inter-microengine communication is shown in Figure 4.9.

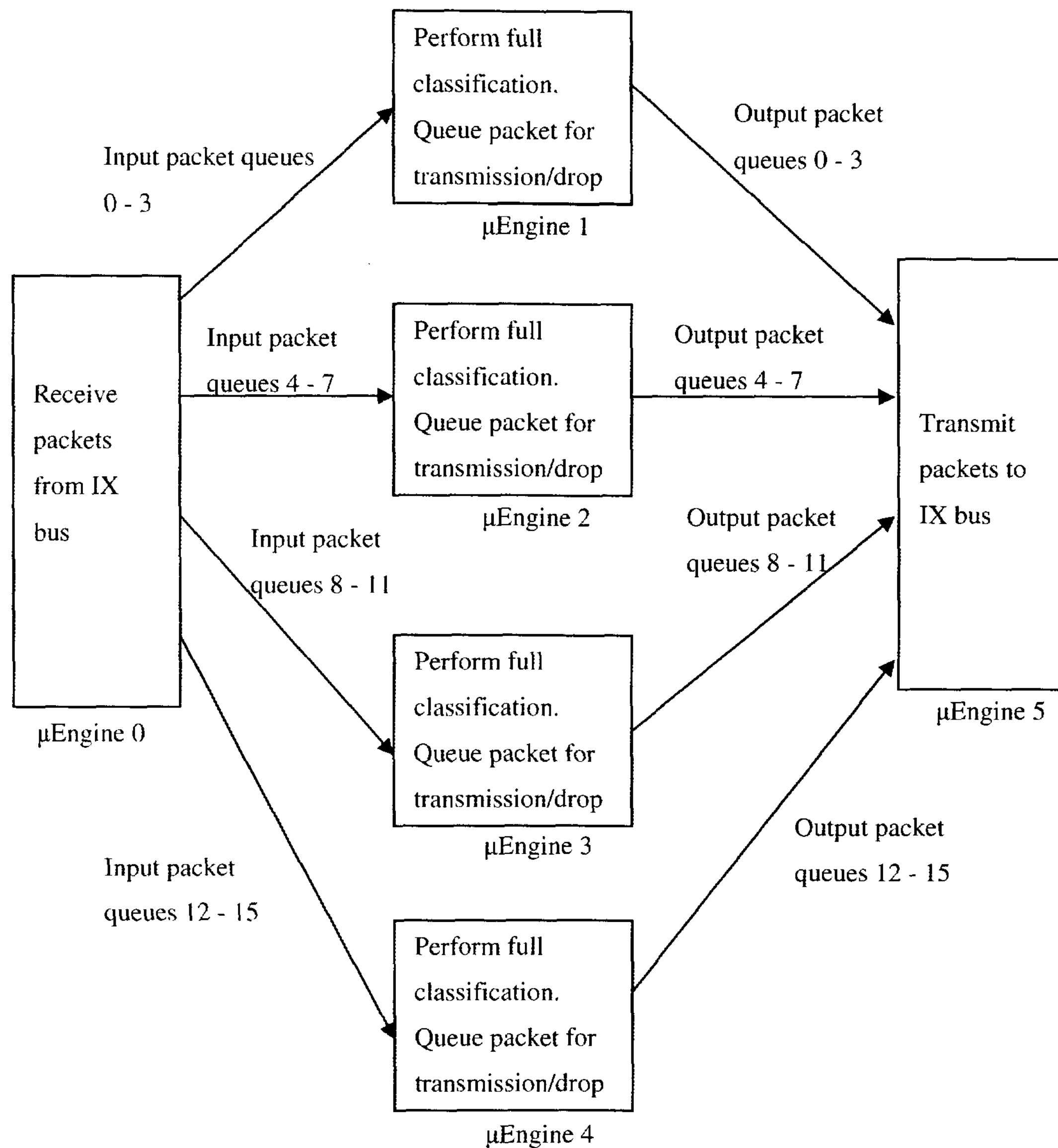

Figure 4.9: Microengine responsibility and communication in BitVector1

4.4.4.2 Implementing BitVector2

As in BitVector1, microengines 0 and 5 perform the *rcv* and *xmit* functions respectively and receive/transmit packets (respectively) from 4 ports. Microengine 0 receives packets from the IX bus and places them in a circular queue in SRAM for use by the *cls* engines. Since in BV2, the classification phase for a single packet is split across different

microengines, the packet goes through a pipeline of microengines. Hence, the *rcv* microengine has to queue packets only for use by the *cls* microengine 1. This is illustrated in Figure 5.10. Microengine 1 performs step 1 of the BV algorithm for a single dimension of a packet (source IP address) and queues it for use by microengine 2. Microengine 2 reads a packet from this queue and performs step 1 for a single dimension of the packet (destination IP address) and queues it for use by microengine 3. Similarly, microengine 3 performs step 1 for the transport layer protocol field/port number and queues it for use by microengine 4.

When each microengine adds the packet to the queue for use by its successor in the pipeline, it also writes the result of its lookup (in the *P*-set) as part of the queue entry. When microengine 4 dequeues a packet from its queue, it can access the results from the previous lookups. It uses these results and performs step 2 of the BV algorithm – it determines the appropriate bit vectors, performs the AND operation and determines the matching rule for the packet. It then performs the action specified by that rule – it queues the packet for transmission by microengine 5 or drops it. Figure 4.10 illustrates in detail the division of responsibility and communication between microengines for BV2.

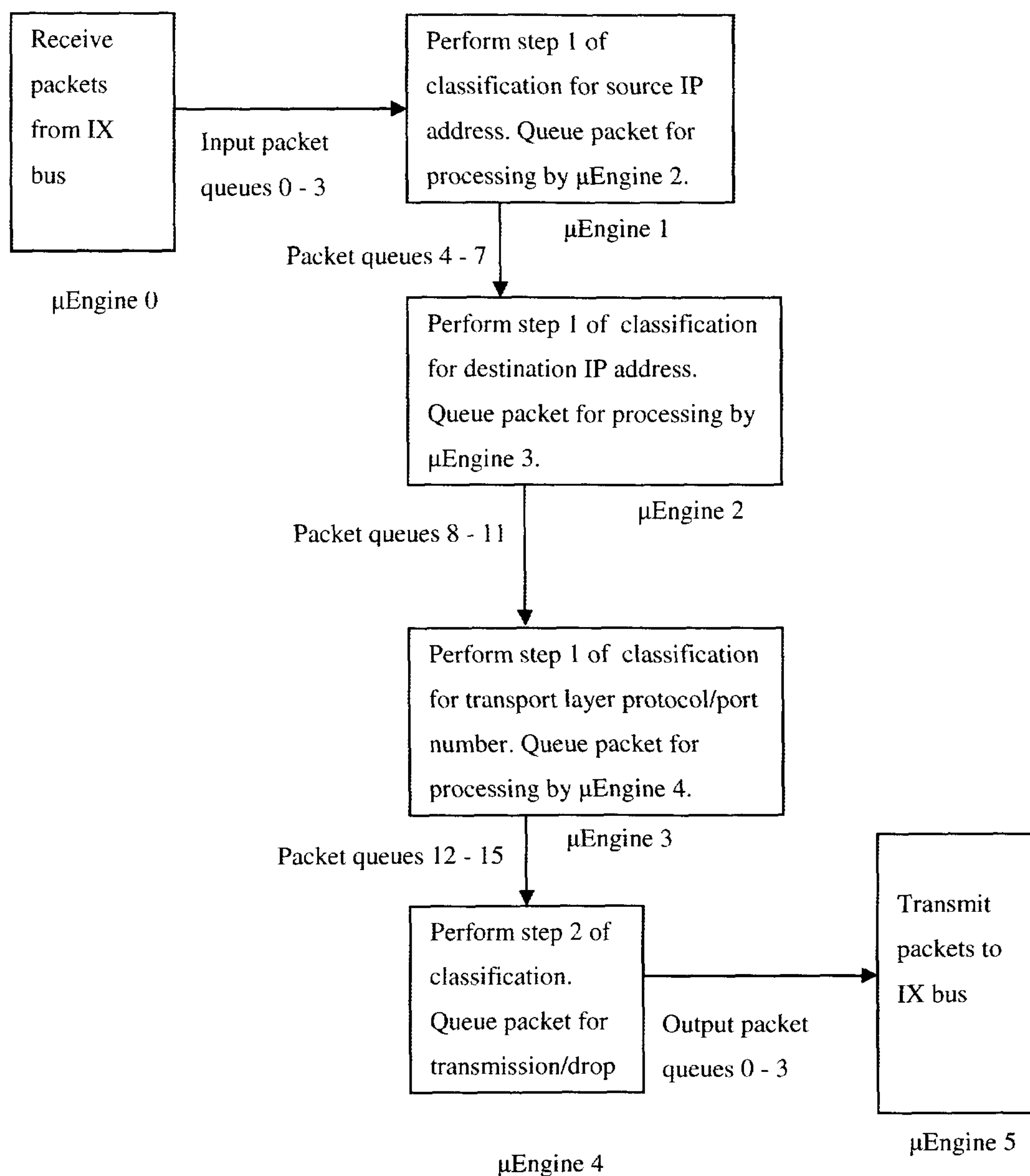

Figure 4.10: Microengine responsibility and communication in BitVector2

4.4.4.3 Data Structure placement in memory

Unlike in GPPs, there is no memory manager on the IXP. Hence, we need to divide the available memory for our use (to store data structures) and determine the exact address in each type of memory that will be used for specific purposes. For example, consider the circular queue in SRAM that is used for inter-microengine communication. The base address of this queue needs to be determined at compile-time, in order to enable different microengines to access the queue. This also implies that care should be taken so as the placement of one data structure in memory does not overlap or interfere with a memory location that is meant for use by a different data structure or for a different purpose. Table 4.1 lists the memory allocation used in BV1 and BV2.

Table 4.1: Memory allocation for BV1 and BV2

Purpose	Type of memory
Queue for inter-microengine communication	SRAM
List of rules actions	SRAM
Tries representing *P-sets*	SDRAM
Bit Vectors	SDRAM

The base addresses of each type of memory also needs to specified at compile time. This is used by the compiler to place variables in memory, as required. Hence, care should be taken so that we specify a base address that is higher than the range of addresses required by our static data structures.

4.4.4.4 Verification of the implementations

After the coding for the different algorithms and design approaches was completed, several tests were run to ensure that the implementation runs as per requirements and to eliminate any defects in the code. This section describes the tests and the results.

Functional tests

We started with a simple development approach – to implement the functionality required, ran it on a single hardware thread on a single microengine and stepped through the code (using the graphical debugger) for a single packet. This level of debugging helped to ensure that the correct code path was being followed for a packet. We first did this process using the *rcv* microengine and a single *cls* microengine. We next included the *xmit* microengine and ensured that a single packet was received, classified and transmitted accurately. We also tested the implementation with some of the rules configured to deny packets and input packets that matched these rules. We verified that such packets get dropped.

The simulator also provides a logging facility, i.e., packets that are transmitted to and from the IXP ports can be logged to individual files. We used this logging feature to ensure that the headers and data in the packets were getting transmitted accurately by the ports.

Once we verified that the basic functionality of the code was working, we increased the number of threads on a single microengine in a systematic fashion and executed the same tests. This helped us detect and debug inter-thread communication issues which caused memory corruption or packets being lost. Once we verified that the code executed properly on one *cls* microengine, we started including the other *cls* microengines in a similar systematic fashion. This helped us detect and debug inter-microengine communication issues which would cause the system to hang, memory corruption, packets being lost etc.

In each of these tests, we also used the logging facility available to ensure that the packets transmitted from the system have accurate headers and data.

Stress tests

After verifying that the implementations worked correctly on all microengines on the IXP, each of the implementations was subject to stress tests. Each implementation was run continuously in the simulator for around 8 hours and we verified that the simulator did not crash and that packets were received and transmitted at steady rates. This helped us debug through issues such as system freeze due to incorrect buffer management and insufficient buffer space.

After these tests were run to verify that the implementations were correct, we executed them again in the simulator to collect performance statistics. These results are presented and analyzed in Chapter 5.

4.5 Other considerations

4.5.1 Management application

This thesis studies only the performance of the algorithm on the microengines only. The microengines are the main components in the IXP that offer the capability to perform wire-speed packet processing. Hence, the performance of an algorithm on the microengines is the important aspect of the overall performance of the algorithm in the IXP system. However, when implementing for real-world use, there is the existence of a management application that typically runs on the main processor of the host system. The presence of such an application has implications for the overall performance of the IXP system since there will be contention to shared data structures in memory. This thesis does not consider the performance implications due to the introduction of a management application or any other source of updates to data structures in memory.

4.5.2 Simulator

The algorithm is implemented in microC and tested and run in the IXP Developer Workbench which offers a cycle-accurate simulator of the IXP. This environment

provides access to several performance metrics that reflect the actual IXP hardware. The code implemented for this thesis can be made to run on the actual IXP hardware. However, running microC microACE code on the hardware is not supported by Intel and hence the process may have some tedium.

4.5.3 Scalability of the Bit Vector algorithm

The size of rulesets used for packet classification varies dramatically depending on its purpose. For example, corporate intranets have approximately 150 rules, whereas large ISPs may have around 2400 rules [5]. The Bit Vector algorithm is well-suited for medium sized rulesets, with around 512 rules [1]. From preliminary study of the basic Bit Vector algorithm, it is evident that it does not scale well for large rulesets due to the large memory requirement for such rulesets. Several optimizations have been proposed [1,7] to the basic algorithm that enable more efficient use of the data structures in memory. This thesis deals only with the basic Bit Vector algorithm since it studies the behavior and performance of different approaches of a particular algorithm. Also, the same fundamental idea of parallelism is present in the extended versions of the algorithm. Since this does not affect the performance evaluation in this study, we do not attempt to modify the basic algorithm to achieve better scalability.

5 Results and Analysis

This chapter presents the performance data from executing the linear search and Bit Vector algorithms on the IXP1200. The data was collected using the IXP1200 Developer Workbench which includes a cycle-accurate simulator. We first present basic performance data using the ruleset given in Table 4.1 with one modification – all the rules have the action set to "Allow", to measure worst-case performance. Using this ruleset, we have $d = 3$, N = 4 and W = 4. (For typical real world rulesets, d ranges from 1 to 5; N ranges from 100s to 1000s; W takes values 4 (for port numbers) and 128 (for IP addresses)).

Simulator configuration

The IXP1200 Developer Workbench allows the user to specify different system configuration parameters that is used by the simulator. For the experiments in this thesis, we use the basic configuration available – we use the default configuration of an IXP1200 chip with 1K microstore that has a core speed frequency of 165.890 MHz. We can also specify configuration settings for the IX Bus Device simulator which controls how packets are sent and received from the simulator. For use in our experiments, we choose a device with 8 ports with a data rate of 100 Mbps and a receive and transmit buffer size of 256. However, since we have only one microengine performing the receive operation, we support only four ports. Hence, we configure the simulator to send packet streams to only ports 0 through 3 of the device.

Experiments setup

To compare the performance of the different mappings, we run each of the implementations in the simulator, until 75000 packets have been received by the IXP from the bus. We then record the various performance metrics and use them for our analysis.

5.1 Performance Results

This section presents the performance results from executing the linear search and bit vector algorithms using the ruleset in Table 4.1. Sections 5.1.1 through 5.1.3 present the detailed performance statistics from each execution. Section 5.1.4 presents a comparison between the three. Section 5.1.5 presents a detailed analysis. Table 6.1 describes the various performance metrics collected for our analysis.

Table 5.1: Performance Metrics Collected

Performance Metric	Description
Microengine cycles	The total number of microengine cycles that was spent to process a specific number of packets.
IX Bus cycles	The total number of IX bus cycles that was spent to process a specific number of packets.
Packets received	The total number of packets that was received by the IXP from the IX bus, to all the ports that are configured to receive packets.
Packets sent	The total number of packets that was sent by the IXP to the IX bus, from all the ports that are configured to transmit packets.
Receive rate (Mbps)	The overall packet receive rate of the IXP, for all the ports that are configured to receive packets.
Transmit rate (Mbps)	The overall packet transmit rate of the IXP, for all the ports that are configured to send packets.
Microengine execution time (%)	The percentage of the total number of microengine cycles that a microengine spent in performing useful tasks.
Microengine aborted time (%)	The percentage of the total time of a microengine that was wasted due to instructions in its pipeline being aborted, typically due to branch instructions.

Microengine idle time (%)	The percentage of the total time of a microengine that was wasted due to none of the 4 hardware threads being available to run, typically due to memory access wait time.
Instruction execution speed (Mips)	The execution speed of a microengine.
SDRAM access (%)	The total percentage of SDRAM bandwidth utilized by all microengines.
SDRAM access rate (Mbps)	The total SDRAM bandwidth used by all microengines.
SRAM access (%)	The total percentage of SRAM bandwidth utilized by all microengines.
SRAM access rate (Mbps)	The total SRAM bandwidth used by all microengines.

5.1.1 Linear Search (LS)

This section presents the performance statistics collected after running the linear search algorithm for the ruleset presented in Table 4.1. Table 5.2 presents the overall performance statistics; Table 5.3 presents the performance statistics for individual microengines.

Table 5.2: Overall performance statistics

Microengine cycles	23,277,945
IX Bus cycles	9,311,178
Packets received	75000
Packets sent	31521
Receive rate (Mbps)	325.35
Transmit rate (Mbps)	136.74
SDRAM access	10.30%
SDRAM access rate	546.8 Mbps
SRAM access	99.60%
SRAM access rate	2645.7 Mbps

Table 5.3: Performance statistics of individual microengines

Microengine	Executing (% time)	Aborted (% time)	Idle (% time)	Mips
0 (rcv)	76.5	20.8	1.3	127
1 (cls)	30.3	12	1.7	50.3
2 (cls)	30.4	12	1.6	50.5
3 (cls)	30.3	12	1.6	50.3
4 (cls)	30.2	11.9	1.6	50.2
5 (xmit)	64.8	13.3	8.2	107.6

5.1.2 Bit Vector Approach 1 (BV1)

This section presents the performance statistics collected after running the Bit Vector algorithm approach 1 (described in Section 4.4.2) for the ruleset presented in Table 4.1. Table 5.4 presents the overall performance statistics; Table 5.5 presents the performance statistics for individual microengines.

Table 5.4: Overall performance statistics

Microengine cycles	22,608,314
IX Bus cycles	9,045,331
Packets received	75000
Packets sent	60643
Receive rate (Mbps)	335.36
Transmit rate (Mbps)	272.5
SDRAM access	41.90%
SDRAM access rate	2228.9 Mbps
SRAM access	46.70%
SRAM access rate	1239.8 Mbps

Table 5.5: Performance statistics of individual microengines

Microengine	Executing (% time)	Aborted (% time)	Idle (% time)	Mips
0 (rcv)	78.4	21.2	0.1	131
1(cls)	66.3	22.5	10.8	111
2 (cls)	66.2	22.5	10.7	111
3 (cls)	66.3	22.5	10.7	111
4 (cls)	66.2	22.5	10.8	111
5 (xmit)	82.4	15.2	2.2	137

5.1.3 Bit Vector Approach 2 (BV2)

This section presents the performance statistics collected after running the Bit Vector algorithm approach 1 (described in Section 4.4.3) for the ruleset presented in Table 4.1. Table 5.6 presents the overall performance statistics; Table 5.7 presents the performance statistics for individual microengines.

Table 5.6: Overall performance statistics

Microengine cycles	19,310,500
IX Bus cycles	7,724,200
Packets received	75000
Packets sent	46343
Receive rate (Mbps)	392.2
Transmit rate (Mbps)	242.34
SDRAM access	45.1
SDRAM access rate	2398 Mbps
SRAM access	47
SRAM access rate	1247.4 Mbps

Table 5.7: Performance statistics of individual microengines

Microengine	Executing (% time)	Aborted (% time)	Idle (% time)	Mips
0 (rcv)	76	23.7	0.4	126.1
1(cls)	55.1	18.4	26.6	91.4
2 (cls)	52.5	17.4	30	87.2
3 (cls)	57.2	18.8	24	94.9
4 (cls - step 2)	68.6	14.5	16.9	113.9
5 (xmit)	82.4	15.4	2.1	136.8

5.1.4 Performance Comparison

This section presents graphical comparisons of the performance metrics of the algorithms presented in Sections 5.1.1 through 5.1.3.

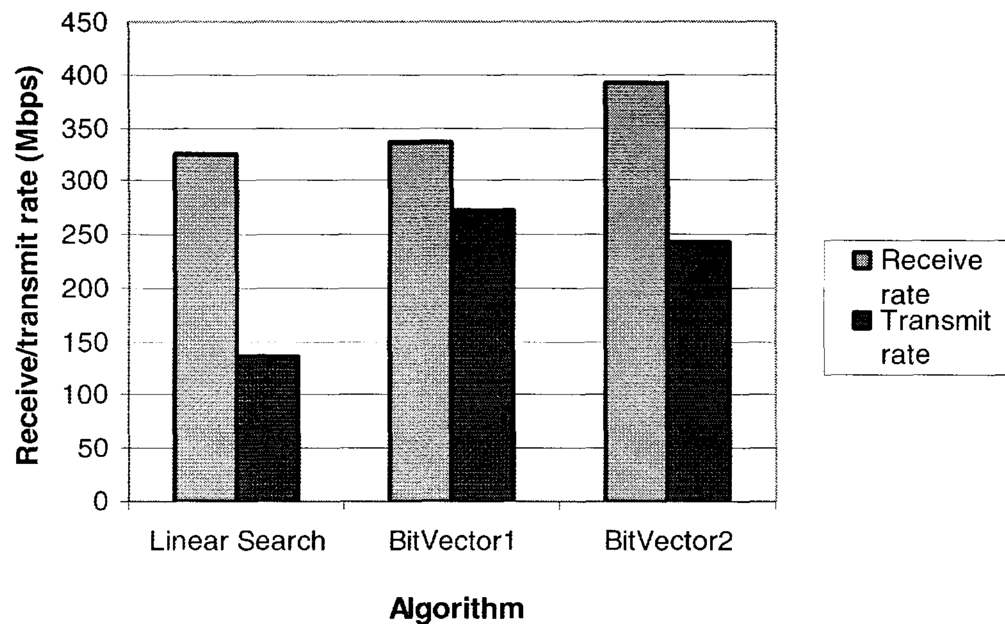

Figure 5.1: Comparison of the receive and transmit rates of the algorithms

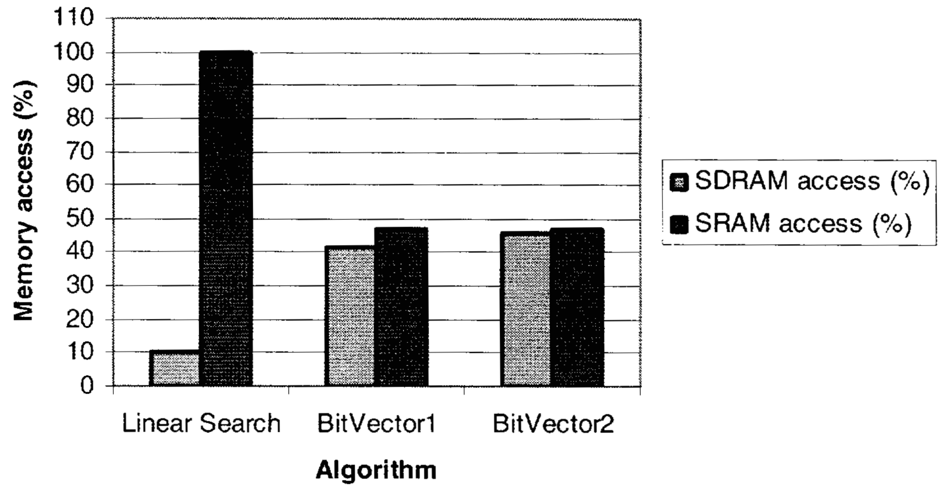

Figure 5.2: Comparison of memory access (%) of the algorithms

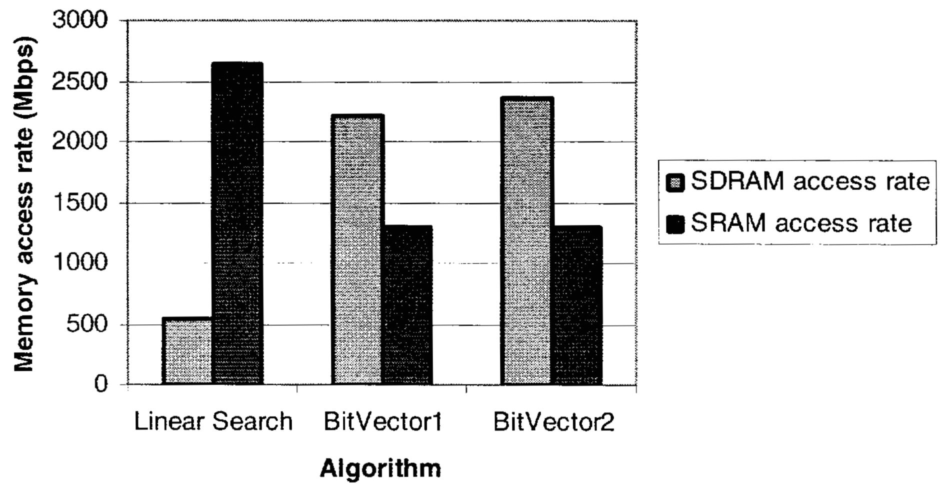

Figure 5.3: Comparison of memory access rates (Mbps) of the algorithms

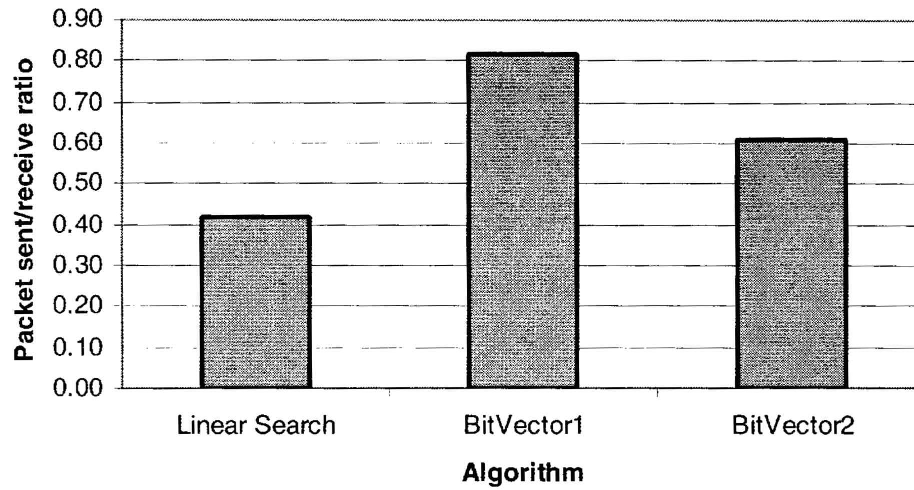

Figure 5.4: Comparison of packets sent/receive ratio of the algorithms

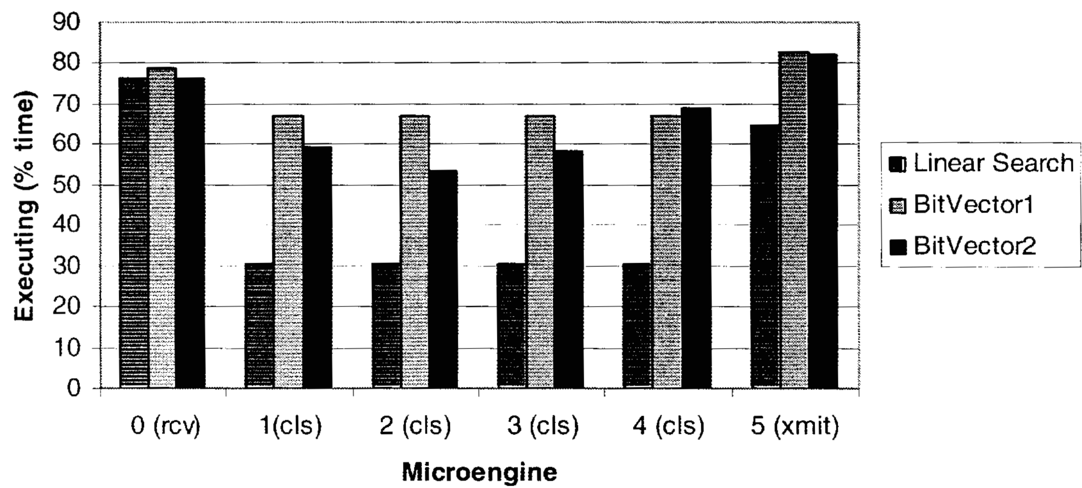

Figure 5.5: Comparison of microengines executing time

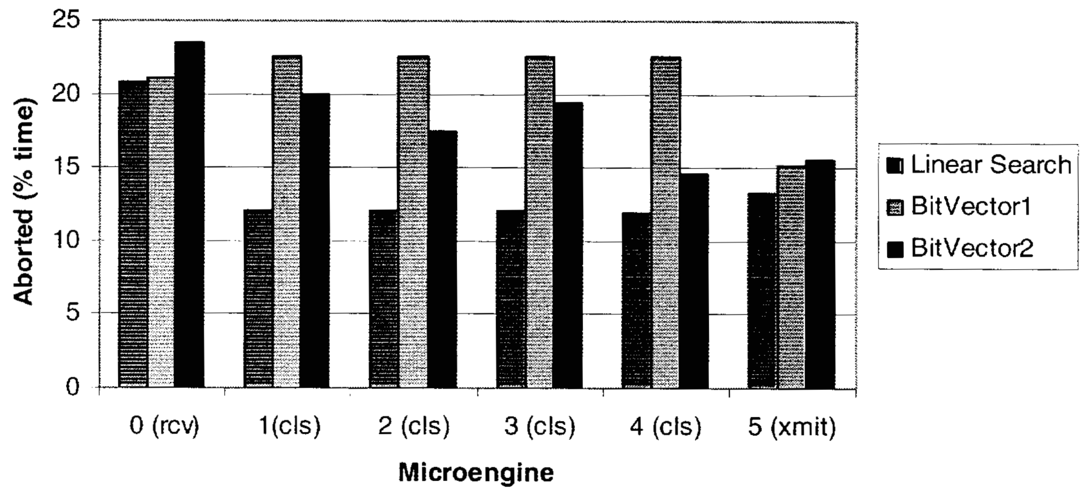

Figure 5.6: Comparison of microengines aborted time

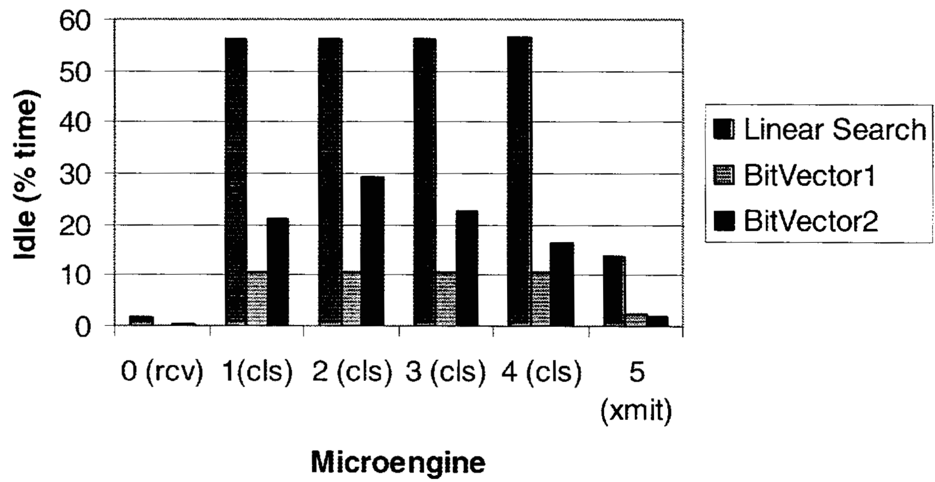

Figure 5.7: Comparison of microengines idle time

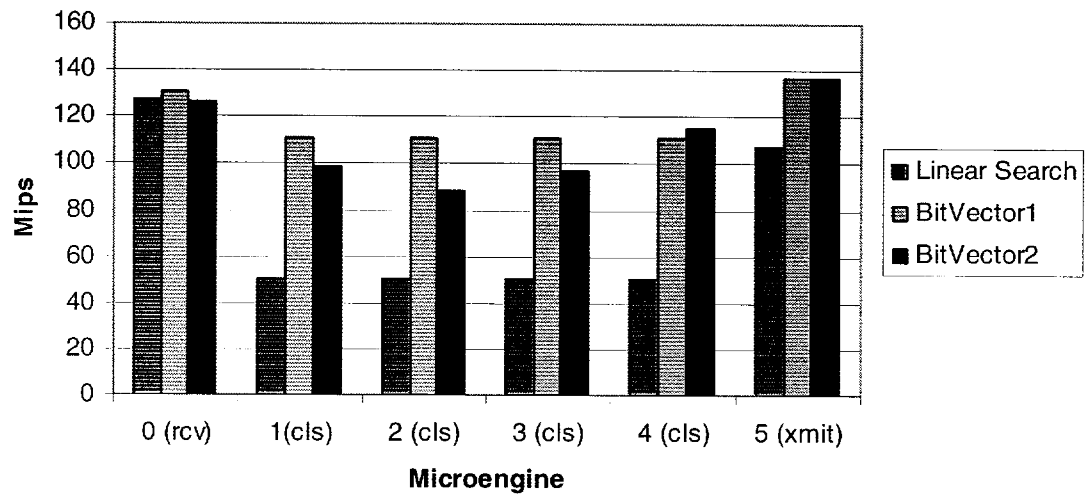

Figure 5.8: Comparison of microengines execution in Mips

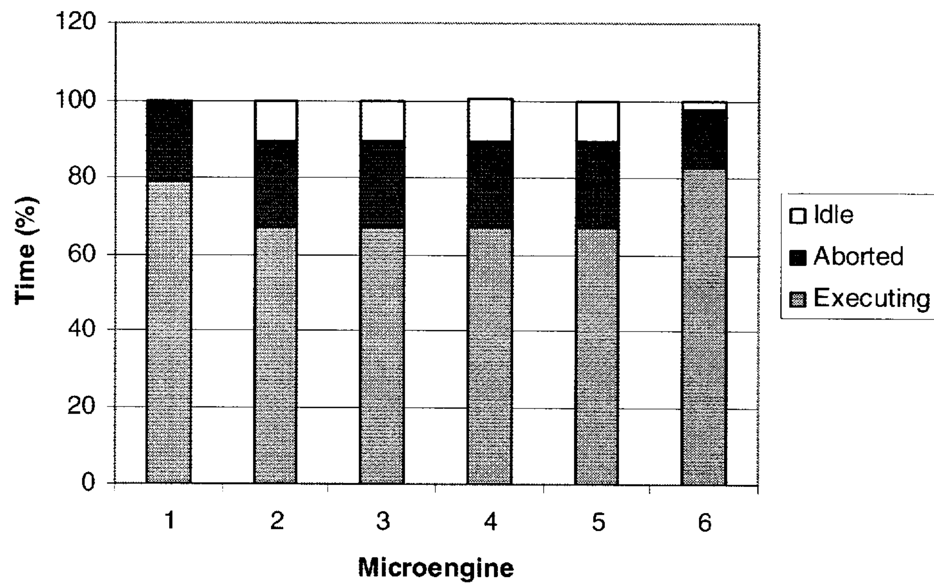

Figure 5.9: Distribution of individual microengine time for BitVector1

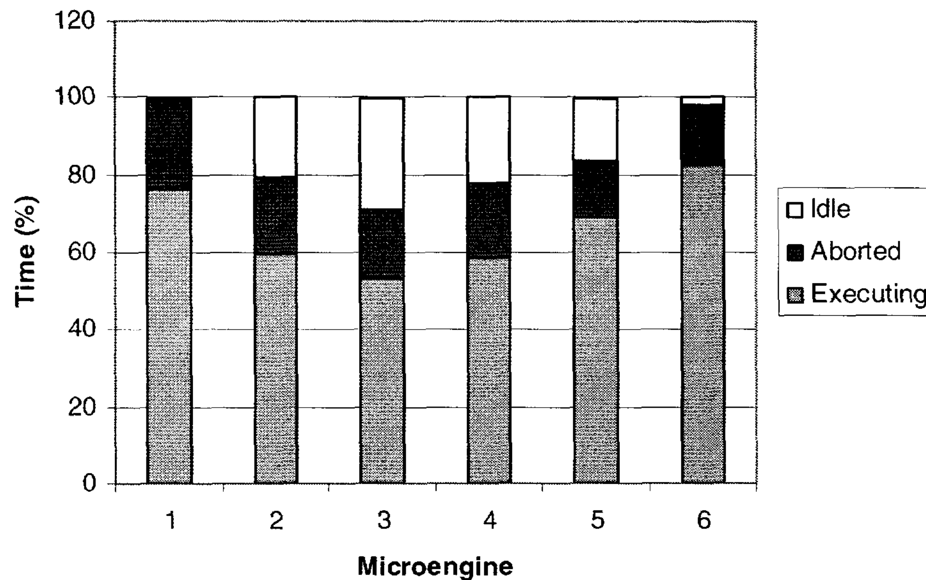

Figure 5.10: Distribution of individual microengine time for BitVector2

5.2 Analysis

This section presents a detailed analysis of the various performance metrics presented in Section 5.1.

5.2.1 Linear Search vs. Bit Vector

This section compares the various performance metrics of the Linear Search algorithm (LS) with that of the Bit Vector algorithm (BV1 and BV2).

Transmit rate

As expected, both the implementations of the Bit Vector algorithm perform significantly better than the Linear Search, since the LS algorithm employs a sequential search through linked list of rules and each access to a rule is a memory (SRAM) access. From Figure 5.1 we can see that the transmit rate of the LS algorithm is significantly lower than that of BV1 and BV2. Given this, we would expect the packet-processing efficiency of the LS

algorithm to be lower than that of BV1 and BV2. This is evident from Figure 5.4 which shows the ratio of the number of packets transmitted by number of packets received.

Memory access

In each of the implementations, the rulesets are stored in SRAM memory and the packets are stored in SDRAM memory. From Figure 5.2, we can see that the LS algorithm's SRAM access rate is significantly higher than that of BV1 and BV2, indicating that it spends a significant portion of the execution time in searching the ruleset. Hence, the number of packets processed per unit time is lower than that of BV1 and BV2. Given this, we would expect the SDRAM access of LS to be significantly lower than that of BV1 and BV2. Figure 5.2 also confirms this hypothesis.

Microengine utilization

Another important performance metric for the IXP is the microengine utilization. The microengines are responsible for fast path packet processing and hence it is imperative that their utilization is as close to 100% as possible. The utilizations of the receive microengines are comparable in LS, BV1 and BV2; the utilization of the transmit microengine depends on the packet classification speed which is handled by the classification microengines. In general, the code executing on the receive and transmit microengines is independent of the main classification algorithm. Hence, we now compare the utilization of the classification microengines (1, 2, 3, and 4) in the LS algorithm and BV1 and BV2. From Figure 5.5, we can see that the execution time of the classification microengines (cls) for LS is typically 50% less than that of BV1 or BV2. Thus, the cls microengines are under-utilized in LS, as would be expected because of the high SRAM access rate.

Microengine aborted time represents the percentage of time an instruction in a microengine pipeline was aborted, typically due to a branch instruction. While the performance of LS is expected to be worse than that of BV1 or BV2, it is interesting to

observe, from Figure 5.6 that the microengine aborted percentage time is significantly less than that of BV1 or BV2. This is because the LS algorithm is simpler in nature and has fewer branch instructions. As we will see in Section 6.2.2, the microengine aborted time for BV1 and BV2 is significant, thus taking away time from the microengine that could have been used for execution. This implies that, although the overall performance of LS is lower than that of BV1 or BV2, in general, simpler algorithms will have lower microengine aborted time. Since this is a significant metric, one design goal that should be adhered to, while designing classification or other class of algorithms for network processors, is that the algorithm should consist of as few branch conditions as possible. This is even more important on the IXP, since each of the four hardware threads on a single microengine share the same code base. Any penalty from a thread being aborted due to branch decision will be multiplied by four.

Although LS has a lower microengine aborted time, the microengine execution time is lower than that of BV1 and BV2 because LS has higher microengine idle time, as would be expected because LS has higher SRAM memory access rate.

5.2.2 BitVector1 vs. BitVector2

This section presents a comparison of the two design approaches of the Bit Vector algorithm's implementation (described in Sections 4.4.2 and 4.4.3) on the IXP. This is one of the main contributions of this thesis since it demonstrates that the packet processing speed of an algorithm on a network processor is dependent on the way it is mapped onto the hardware.

While comparing the performances of these two design approaches, it is important to keep in mind the allocation of the microengines for BV1 and BV2:

- In both BV1 and BV2, microengines 0 and 5 perform the receive and transmit functions respectively.
- In BV1, microengines 1, 2, 3 and 4 perform the full classification functions.

- In BV2, microengines 1 and 2 perform lookup for IP addresses; microengine 3 performs lookup for the transport layer port number (or protocol); microengine 4 performs the step 2 of the BV algorithm – it combines the results from the previous lookups to determine the matching rule.

Whereas Section 5.2.1 presented separate analyses of individual performance metrics, the same cannot be done while comparing BV1 and BV2. This is because, as we will see below, in this case, the analysis is dependent on considering multiple performance metrics at one time. Hence, we present below an overall analysis of BV1 and BV2. As an exception, we note that it is interesting to compare the microengine aborted time separately, and that is presented first.

Microengine Utilization

From Figure 5.6, we see that the aborted time for the classification engines is lower in BV2 than that of BV1. In particular, the aborted time for microengine 4 (which performs the full classification in BV1 and only step 2 of the classification in BV2) in BV2 is approximately 60% that of BV1. This is because in step 2 of the Bit Vector algorithm, we perform simple operations such as reading the bit vector from memory and performing an AND operation. Since microengine aborted time is typically because of branch instructions, microengine 4 of BV2 exhibits a significantly lower aborted time than in BV1. This emphasizes the conclusion that choosing simpler algorithms or designing algorithms that use lesser branch instructions is important in improving microengine utilization.

The overall microengine utilization time is lower in BV2 than in BV1. This is because the microengine idle time (i.e., excluding microengine aborted time) is higher in BV2 than BV1. In BV2, the processing time required for a packet is higher than that of BV1. Hence, although a microengine may have completed its portion of the processing for a packet, due to the buffering that occurs between microengines, microengines 1, 2 and 3

exhibit significantly higher idle time. That is, in BV2, a new packet cannot be handled by microengines earlier in a pipeline until there are available inter-microengine buffer entries; these entries are freed only when the entire processing for that packet is completed by all microengines and the packet has been queued for transmission, by microengine 4.

Overall Analysis

In BV2, we split the various steps in processing a packet, across microengines. The parallel nature of the BV algorithm is well-suited for this kind of division of responsibility on parallel hardware. However, we observe that the performance (in terms of packet transmit rate and packet sent/receive ratio) of BV2 is lower than that of BV1. The execution time of the classification microengines in BV2 is comparable to those in BV1. However, the packet processing rate of BV2 is lower than BV1. This is because: in BV1, the SDRAM access to read the packet header for classification occurs only once, by a single hardware thread of the microengine that is performing the entire classification for that packet; in BV2, splitting the lookups in step 1 of the BV algorithm across microengines for a single packet, causes three hardware threads on different microengines(1, 2 and 3) to access the packet header in SDRAM for that packet, thus increasing the memory access time required to process one packet by three times. Hence, the speed of packet processing is reduced by 25% in BV2 than in BV1 (seen in Figure 6.4)

6 Conclusion

There are several packet classification algorithms that exist in current literature. However, they are not immediately applicable to programmable network processors such as the Intel IXP1200. The highly parallel nature of the hardware architecture of the IXP seems to be suitable for a parallel search packet classification algorithm. Hence, we study the performance and behavior of the Bit Vector algorithm, which is an example of a parallel search algorithm that is suitable for implementation in parallel hardware. We have proposed different design mappings of this algorithm on the IXP and observed how different design mappings of the algorithm onto the IXP hardware affect its performance. This is important since, while actually implementing an algorithm on the IXP, we deal with details such as buffer management, queuing, memory accesses and potential bottlenecks in the implementation. Thus, given an algorithm, choosing the best possible mapping of the algorithm onto the IXP is critical for achieving optimal performance.

This thesis has presented performance results and analysis from running a linear search algorithm and two different design approaches of the Bit Vector algorithm on the IXP hardware. While BV2 seems superior in design to BV1 since it splits the packet classification for a single packet across microengines, on the IXP1200, we have seen that this results in lower utilization of the microengines. Hence, BV1 has a higher transmission rate than BV2. We also compare the performance of the Bit Vector algorithm with that of a Linear Search and see that, as expected, the former has superior performance. From the performance metrics, we also see that an important performance metric to be considered is the microengine aborted time, which occurs typically due to branch instructions in the code. In both the BitVector approaches, we note that thread aborted time for individual microengines ranges from 14% - 23%. This indicates that algorithms which have frequent complex branch decisions will perform worse than those that have simpler execution paths.

This thesis is an attempt to study in detail the performance of an important class of algorithms on programmable network processors. It has presented insights into how the mapping of an algorithm on a network processor may affect its performance. These results can be used while designing packet classification algorithms or other class of algorithms for implementation on programmable network processors.

References

[1] T.V. Lakshman, D. Stiliadis. "High-speed policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching". Proceedings of ACM Sigcomm, Volume 28 Issue 4, pages 191-202, October 1998.

[2] P. Gupta and N. McKeown. "Algorithms for Packet Classification". IEEE Network Magazine, Volume 15 Issue 2, pages 24 – 32, March/April 2001.

[3] Intel Corporation. "IXP1200 Hardware Reference Manual". Intel IXP1200 Software Developers Kit 2.0.

[4] E. Johnson and A. Kunze. "IXP1200 Programming", published by Intel Press, ISBN 0-9702846-7-5, 2002.

[5] M. Kounavis, A. Kumar, H. Vin, R. Yavatkar and A. Campbell. "Directions in Packet Classification for Network Processors", Second Workshop on Network Processors (NP2), Anaheim, California, February 8-9 2003.

[6] V. Srinivasan, G.Varghese, S.Suri and M. Waldvogel. "Fast and Scalable Layer Four Switching". ACM SIGCOMM Computer Communication Review, Volume 28 Issue 4, pages 191-202, October 1998.

[7] M.M. Buddhikot, S. Suri, M. Waldvogel. "Space Decomposition Techniques for Fast Layer-4 Switching". Sixth International Workshop on Protocols for High Speed Networks, Salem, Massachusetts, August 25-27 1999.

[8] A. Feldmann and S. Muthukrishnan. "Tradeoffs for Packet Classification". Proceedings of Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Volume 3, pages 1193-1202, March 2000.

[9] P. Gupta and N. McKeown. "Packet Classification on Multiple Fields". Proceedings Sigcomm, Computer Communication Review, Volume 29, Issue 4, pages 147-160, September 1999.

[10] P. Gupta and N. McKeown. "Packet Classification using Hierarchical Intelligent Cuttings". Proceedings of Hot Interconnects VII, August 99, Stanford University. This

paper is also available in IEEE Micro, Volume 20, Issue 1, pages 34-41, January/February 2000.

[11] F. Baboescu and G. Varghese. "Scalable Packet Classification". ACM Sigcomm, San Diego, California, August 27 - 31 2001.

[12] C. Partridge. "Gigabit Networking", published by Addison-Wesley. ISBN: 0-201-56333-9, 1993

[13] Gigabit Ethernet Alliance. "Gigabit Ethernet White Paper". http://www.10gea.org, 2003.

[14] C. Partridge et al. A 50- Gb/s IP Router. IEEE/ACM Transactions on Networking, Volume 6 Issue 3, pages 237-248, June 1998.

[15] Intel Corporation. Intel Network Processors product information. http://www.intel.com/design/network/products/npfamily, 2003.

[16] IBM Corporation. IBM PowerNP product information. http://www-3.ibm.com/chips/products/wired/products/network_processors.html, 2003.

[17] P.N. Glaskowsky. "Intel beefs up networking line". Article reprint from Microprocessor Report, March 2002, available at http://www.intel.com/design/network/papers/Intel_Ntwkg_Rprnt.htm, 2002.

[18] T. Spalink, S. Karlin, L. Peterson. "Evaluating network processors in IP forwarding". Princeton University Technical report TR-626-00, November 2000.

[19] Intel Corporation. "Intel Microengine C Compiler Language Support Reference Manual", Intel IXP1200 Software Developers Kit 2.0. March 2002.

Biographical Sketch

Deepa Srinivasan was born on 6th February 1977 in Chennai, India. She received her Bachelor of Science in Mathematics from the Ethiraj College for Women at the University of Madras in 1997. She also completed a Professional Diploma in Software Technology and Systems Management from the National Institute of Information Technology, Chennai, India in 1997. Her Master of Science program at the OGI School of Science & Engineering was focused in the areas of computer architecture, operating systems, networking and security. She has been working in various software engineering positions in India and USA and presently works as a Staff Software Engineer at IBM Corp.