DESIGN AND EVALUATION OF PACKET CLASSIFICATION SYSTEMS ON MULTI-CORE ARCHITECTURE

By

SHARIFUL HASAN SHAIKOT

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

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

AUGUST 2012

To the Faculty of Washington State University:

The members of the Committeee appointed to examine the dissertation of SHARIFUL HASAN SHAIKOT find it satisfactory and recommend that it be accepted.

Min Sik Kim, Ph.D., Chair

Carl H. Hauser, Ph.D.

David E. Bakken, Ph.D.

ACKNOWLEDGEMENTS

I sincerely thank my research advisor Dr. Min Sik Kim for his tremendous patience and diligent mentoring over the course of the last few years. He taught me the way to see the light at the end of the tunnel and also as to how to conduct research and guided me throughout this work. It is very rewarding to work with Dr. Kim. His sharp intuition and insights on the essence of the problems in the world of research is truly inspiring. My dissertation would not have this complete shape today without the countless discussions I had with Dr. Kim during my stay as Research Assistant in Washington State University.

It has also been an honor to have Dr. Carl H. Hauser and Dr. David E. Bakken in my Ph.D. research committee. I highly appreciate their time for the discussion on several occasions about my research and for providing me the relevant reading materials which really helped me to understand the insight of theories related to my research and for their valuable feedbacks from the early stage to the final output of this dissertation. I also would like to thank both Dr. Carl H. Hauser and Dr. David E. Bakken for serving on my committee.

I would like to extend my thanks to the staff and faculty members of the Department of Computer Science at Washington State University for their numerous and kind assistance during the whole period of my research work at the Department.

Finally, I would like to offer my most heartfelt thanks to my wife and best friend Rupa and my cute little daughter Sabeha for their love, companionship and undying support throughout my Ph.D. program which helped me to focus on my research with undivided attention. I am grateful to my parents for their love and for their inspiration, unconditional support and encouragement throughout my life. I thank my two lovely sisters, two nephews, and the rest of my family for their constant support and also my in-laws family members, especially my mother-in-law, for having their faith and trust on me.

Thank you Ammu, Abbu, Rupa, Sabeha and the rest of my near and dear ones of the family members: without your support and constant encouragement I wouldn't be able to complete this dissertation. I dedicate this dissertation to all of you.

Shariful Hasan Shaikot, Ph.D.

Washington State University,

Pullman, WA, USA

August 2012

DESIGN AND EVALUATION OF PACKET CLASSIFICATION SYSTEMS

ON MULTI-CORE ARCHITECTURE

Abstract

by Shariful Hasan Shaikot, Ph.D.
Washington State University
August 2012

Chair: Min Sik Kim

Packet classification (PC) is the core mechanism used by network devices such as edge routers, firewalls, and intrusion detection systems to classify incoming traffic based on the classification policy. In decision-tree-based PC, packets are classified by searching in tree data structure. However, tree search presents significant challenges because it requires a number of unpredictable and irregular memory accesses. Packet classification is per-packet operation and memory latency is considerably high (caused by cache and TLB misses). The growing trend of number of rules in the classifier coupled with the constant increase in link speeds makes wire-speed classification a challenging task. Hence, satisfactory performance of PC still remains elusive at the wire speed. In this dissertation, we propose several novel ideas to improve the look up performance of packet classification system. They are:

1. An efficient memory layout for the tree data structure which ensures the movement of data optimally among the different levels of the memory hierarchy on modern general purpose processors. In particular, the number of accessed cache lines (and memory pages) is minimized by our proposed memory layout resulting in less number of cache and TLB misses;

2. `npf`, a traffic-adaptive packet classification system which exploits the potentiality of traffic

locality to optimize the average look up time. It dynamically reorganizes the internal data structure in order to adapt to the traffic characteristics. Unlike existing approaches requiring a separate, off-line reorganization phase, `npf` performs reorganization on-line with little overhead, resulting in improved look up time per packet on average;

3. `Pnpf`, a parallel traffic-aware classification system that exploits the strong computational power and thread-level parallelism capabilities of modern multi-core general purpose processors in order to achieve Gbps classification rate;

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER one

INTRODUCTION

1.1 Packet Classification

Today's Internet traffic consists of different types of traffic, such as time sensitive, malicious or benign. Thus, every Internet core and edge router or firewall today needs to treat these traffic mix differently. Packet classification is a technique that is used by these network devices to classify the traffic and process them accordingly. For example, an Intrusion Detection System (IDS) classifies the packet either as benign or malicious based on the policy in the rule-set. The rule-set consists of multiple rules that characterize the pattern of the malicious activity. Individual entries for classifying a packet are called rules. For instance, a typical firewall rule specifies that packets from which subnet should be blocked in order to protect network against malicious attacks. In this context, the packet classification problem is to determine the first matching rule for each incoming packet.

Recent advancement in transmission link rates (Gbps) at the core and edge of the Internet and growing trend of large rule-sets size [4] pose challenges on the existing traditional packet classifiers. Currently the largest rule-set in use contain thousands of rules and each rule involves five or more header fields. Tens of thousands of rule in a rule set are expected in the future. Currently, core routers are connected by OC-768 (40 Gbps) link and edge routers are connected by OC-192 (10 Gbps) link. It is estimated that in worst case scenario, more than 30 million packets need to be classified in a second to perform wire-speed processing in OC-192 i.e. the classification results must be produced every 32 ns. Therefore, packet classification is still an open and challenging problem and any technique that can reduce the lookup time per packet can be useful in practice for fast packet classification.

1.2 Problem Statement of Packet Classification

Table 1.1: An example of a typical rule-set

ruleID	SrcIP	DestIP	SrcPort	DestPort	Protocol	Action
1	*	*	[600,610]	80	TCP	Accept
2	*	*	[1000,1024]	[1024,65535]	TCP	Reject
3	*	*	23	80	TCP	Accept
4	*	66.166.49.30	25	1024	TCP	Reject
5	66.166.49.151	*	22	*	TCP	Accept
6	*	66.166.49.30	*	*	UDP	Reject
7	*	*	[1000,1024]	80	TCP	Accept
8	*	*	*	*	*	Reject

Packet classifier categorizes packets based on a set of pre-defined rules that represent the classification policy. Typically, the values of specific fields in packet headers are used in packet classification. For example, in IPv4, the IP 5-tuples, the source/destination IP addresses, source/destination port numbers and protocol are used for classifying packets. Other header fields, e.g. the TCP flags, can also be used for the classification. The *rule database* or *rule-set* consists of a finite sequence of rules, $R_1, R_2, \ldots, R_N$. Rules in the rule-set contain values for each of the five fields. The rules can be a combination of an *exact match*, *prefix match* or *range match* over various fields of the packet header. The packet header field should:

1. exactly match the rule field in an exact match
2. match with the rule field that is a prefix of the packet header in a prefix match
3. lie within the range specified by the rule field in a range match

For example, exact matching is useful for protocol, prefix matching is useful for taking some action for a certain subnet and range matching is applicable for situation where port number ranges is specified. Each rule R_i is a combination of d values, one for each header field or dimension and has an associated action A_i which decides the fate of the packet. In a typical rule-set, $A = \{\text{accept}, \text{reject}\}$. We say that a packet P matches a rule if all the header fields of P match all the corresponding rule-fields of R. For example, according to Table 1.1, a packet P matches ruleID 1 if the TCP packet is originated from any IP address with source port between 600–610 and is destined to destination port 80 for any IP address. Since any packet may match multiple rules in a typical rule-set, the first matching rule (based on the rule ordering) is given the highest priority. Usually, the rule's position in an ordered list of rules defines its priority. If a packet does not match any of the rules in the rule-set, then the fate of the packet is decided by the default rule which is usually set by the network administrator.

1.3 Solutions of Packet Classification Problem

The most simple solution of packet classification problem is a technique that linearly searches through the rule-set to find a matching rule. However, for large rule-sets that often contain hundreds to thousands of entries, this simple and memory efficient technique is unacceptably slow and classification must be performed at link-speed in order to avoid creating a bottleneck. In order to improve the search times, specialized data structures, geometric algorithms, and heuristics have been proposed or expensive custom hardware such as Ternary Content Addressable Memory (TCAM), Application-specific Integrated Circuit (ASIC), Field-programmable Gate Array (FPGA) has been embraced to tackle the problem. Both of these approaches have their own pros and cons in terms of performance, scalability and cost. Comprehensive surveys of packet classification techniques can be found in [5–7]. There have been two major threads of research addressing the

problem: software-based (algorithmic) solutions and hardware-based (TCAM-based) solutions. In this section, we discuss certain high level characteristics of both threads of solutions.

1.3.1 Hardware-based solutions

Ternary Content Addressable Memories (TCAMs) are special-purpose memory modules which allow three possible values to be stored in a memory cell, i.e. 0, 1, or x (don't care) and can execute fast parallel searches on all of its stored contents simultaneously. TCAMs are the most widely used packet classification technique in high performance network routers because of their deterministic and high-speed lookup (TCAM can perform an IP address lookup in one clock cycle). The high degree of parallelism [5] supported by TCAMs make them effective for using in packet classification. TCAM-based scheme can classify 250 million packets per second, which satisfies the throughput demands of all the existing networks today [8]. The use of TCAMs in packet classification (routing table lookups) was first proposed by McAuley and Francis [9]. While TCAMs remain the most dominant solution for high performance packet classification in network routers, it suffers from several deficiencies [6] compared to commodity hardware: (1) expensive (high cost per bit relative to other memory technologies), (2) storage inefficiency (inefficient representation of range match fields), (3) high power consumption, and (4) limited scalability to long input keys, (5) lack of flexibility and programmability, (6) unsuitable for dynamic rules, since incremental updates usually require many TCAM entries to be shifted. Compared to static random access memories (SRAMs), TCAMs are expensive and do not scale well with respect to clock rate, power consumption, or circuit area with rule-set size. That is why the algorithmic alternatives for general packet classification using SRAM/DRAM are still being considered. A detailed summary of the trade-offs involved between software and TCAM-based approaches can be found in [6, 10].

1.3.2 Software-based solutions

Software-based classification is the most flexible and enables more sophistication and complexity at the cost of slower execution time. Packet classification algorithms use two dominant resources, memory and time. For example, packet classification problem can be mapped to the point location problem where one has to find the enclosing region of a query point, given a set of non-overlapping regions. In the context of packet classification, rules define hyper-cubes and a packet defines a point in the space. The goal is to determine the highest priority hyper-cube covering a given point. Point location in computational geometry has proven to be difficult. Assuming there are `n` rules and `d` dimensions, [11] shows that it is possible to achieve $O(\log n)$ lookup time, but with a complexity of $O(n^d)$ in storage; while optimizing for storage leads to $O(n)$ storage requirements, a lookup may take $O(\log^{d-1} n)$ time to finish. Clearly, both extremes are unacceptable in practice because with just 1000 rules and 3 fields, $O(n^d)$ space is about 1GB; and $O(\log^{d-1} n)$ time is about 100 memory accesses [12].

The Packet classification problem has been extensively studied and consequently numerous algorithms and architectures for packet classification have been proposed. The comprehensive survey [6] presents a taxonomy that breaks the design space into four regions based on the high-level approach to the problem. The high-level approaches to finding the best matching rule or rules for a given packet:

- Exhaustive Search: all rules in the rule-set are searched sequentially
- Decomposition: The multiple field search is decomposed into instances of single field searches, and searches are then performed independently on each packet header field and finally the classification result is determined by combining these independent intermediate results

- Decision Tree: a decision tree is constructed from the rules in the rule-set and the packet header fields are used to traverse the decision tree to find the matching rule
- Tuple Space: the rule-set is partitioned according to the number of specified bits in the rules. The partitions or a subset of the partitions are then probed using simple exact match searches

Crossproducting [13] was one of the first techniques to employ decomposition. The observation that the number of unique field specifications is significantly less than the number of rules in the rule-set motivated the design of Crossproducting. For example, a rule-set containing 100 rules may contain only 17 unique source address prefixes, 10 unique destination address prefixes, 11 unique source port ranges, etc. Classification is done first by constructing a crossproduct table from the rules in the rule-set and then by performing lookup into this table using packet header values. The entries in the direct lookup table (crossproduct table) are all possible field value combinations (cross-products) and the earliest rule matching each cross-product (pre-computed). To locate the best matching rule for a given packet, the approach performs separate lookups on each field independently, concatenate the results from the independent lookups and use this concatenated string to index into the crossproduct table.

Recursive Flow Classification (RFC) [5] is a variation of the crossproducting algorithm. This heuristic based approach provides high lookup rates at the cost of memory inefficiency. The authors introduced a unique high-level view of the packet classification problem in which they showed that the packet classification can be viewed as the reduction of an m-bit string defined by the packet header fields to a k-bit string specifying the set of matching rules for the packet. For classification on the IPv4 5-tuple, m is 104 bits and k is typically on the order of 10 bits. Similar to the Crossproducting technique, RFC performs independent, parallel searches on "chunks" of the packet header, where "chunks" may or may not correspond to packet header fields. The results of

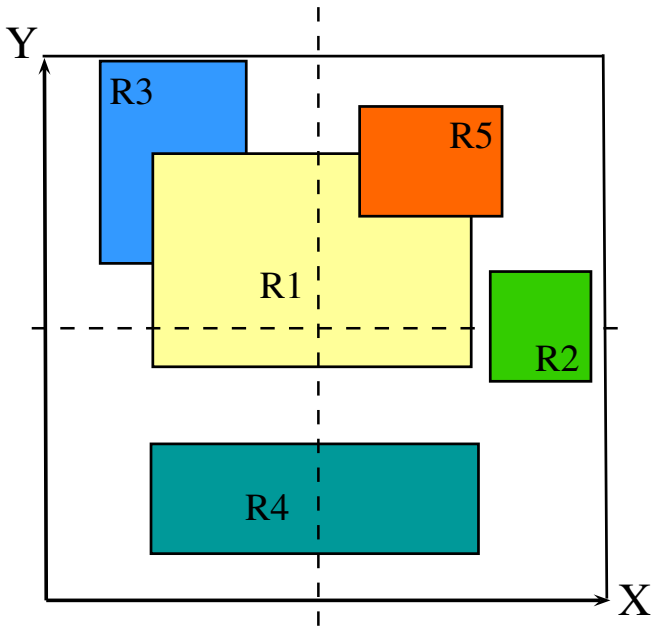


Figure 1.1: Recursive cutting in 2-Dimenstion

the "chunks" searches are combined in multiple phases, rather than in a single step as is done in Crossproducting. The result of each "chunks" lookup and aggregation step in RFC is an equivalence class identifier, eqID, that represents the set of potentially matching rules for the packet.

By looking at the packet classification problem from geometric view, researchers gain new insight and ideas on how to construct the data structures and to represent packet rules. In the geometric view, many algorithms cut the search space recursively into smaller subregions. Each subregion contains fewer rules. This recursive cutting procedure narrows down the search space. Figure 1.1 illustrates recursive cutting on a 2D plane. The decision tree-based algorithms usually apply the cutting technique (Chapter 2).

Most of the existing packet classification algorithms reported in the literature exploits the characteristics of rule-set in optimizing their techniques. The first technique is rule-set partitioning. It uses a few packet header bits to partition the rule-set space into a set of equisized smaller

subspaces which distribute the rules as evenly as possible. Some other header bits are then used to continue partitioning each subspace. This recursive partitioning scheme results into a decision tree. Software-based solutions using this approach include Woo's modular packet classification [14] , Hierarchical Intelligent Cuttings (HiCuts) [12], Multidimensional Cuttings (HyperCuts) [4].

The second technique is rule-set intersecting. In this approach, instead of matching the entire rule at one time, partial matching to a rule is done. The packet header field is split into a set of substrings and then each substring can match a subset of rules. The rules that match the full packet header is obtained by taking the intersection of these subsets. The Bit Vector (BV) algorithm [15] and the Aggregated Bit Vector (ABV) algorithm [16] explicitly represent the subset of rules for each partial match by using bitmap. BV treats each dimension independently. On classification, d (assuming d-dimensions) parallel lookups are issued, each yielding a n bit vector stating the matching rules for the corresponding dimension. The vectors are then intersected by applying bit-wise AND operations and the resulting bit vector is further processed like the TCAM bit vector. The aggregated bit vector (ABV) algorithm, extension of BV, assumes sparse vectors, i.e. vectors with a very small number of 1s, to reduce both memory usage and number of parallel operations. These algorithms work well for moderately size rule-set because the inherent linear worst case scaling makes it difficult to scale up to large rule-set. The intersection operation can be done in sequence or in parallel. For example, the RFC algorithm [5] and the Distributed Cross-producting of Field Labels (DCFL) algorithm [17] employ parallel set intersection in multiple steps in a recursive manner. The Fat Inverted Segment Trees (FIST) algorithm [18] uses a similar approach but performs the intersections in sequence. The partial header lookup can be done using different methods. The simple and the fastest method is to use a direct lookup table. However, it is not scalable in terms of storage consumption. Any single field lookup technique, such as binary search and longest prefix matching can also be used.

The last technique to exploit rule-set characteristics is rule-set grouping. Some rules are regrouped into disjoint subsets based on the common feature that they share. Parallel search is executed on each of these smaller subsets. The results of all the searches is used to determine the best matching rule. This idea is proposed in Tuple Space Search [19], in which rules are grouped based on a tuple specification. A simple hash table is used to support lookups (simple exact match searches) in each tuple.

Each of these approaches either exploit the characteristics of rule-set in their optimization techniques or try to optimize their technique for the worst case scenario by minimizing the depth of the search tree (Hi-Cuts [12] and HyperCuts [4]). However, we pursue a different avenue in our optimization technique. We propose a classification system that exploits the potentiality of traffic locality to optimize the average look up time. Our study on publicly available traffic traces [3] and traffic traces captured in our research laboratory reveals that the majority of the incoming (or outgoing) packet is matched against a small subset of rules in the rule-set (skewness of the traffic matching the rules) and this traffic skewness property stays over an extended period of time. Our findings are consistent with the prior observations [2, 10, 12, 20–22] as well.

1.3.3 Our Contributions

The key contributions in this dissertation are as follows:

1. An efficient memory layout for the tree data structure which ensures the movement of data optimally among the different levels of the memory hierarchy on modern general purpose processors. In particular, the number of accessed cache lines (and memory pages) is minimized by our proposed memory layout resulting in less number of cache and TLB misses (Chapter 3);

2. `npf`, a traffic-adaptive packet classification system which exploits the potentiality of traffic locality to optimize the average look up time. It dynamically reorganizes the internal data structure in order to adapt to the traffic characteristics. Unlike existing approaches requiring a separate, off-line reorganization phase, `npf` performs reorganization on-line with little overhead, resulting in improved look up time per packet on average (Chapter 4);

3. `Pnpf`, a parallel traffic-aware classification system that exploits the strong computational power and thread-level parallelism capabilities of modern multi-core general purpose processors in order to achieve Gbps classification rate (Chapter 5).

CHAPTER two

DECISION TREE BASED PACKET CLASSIFICATION

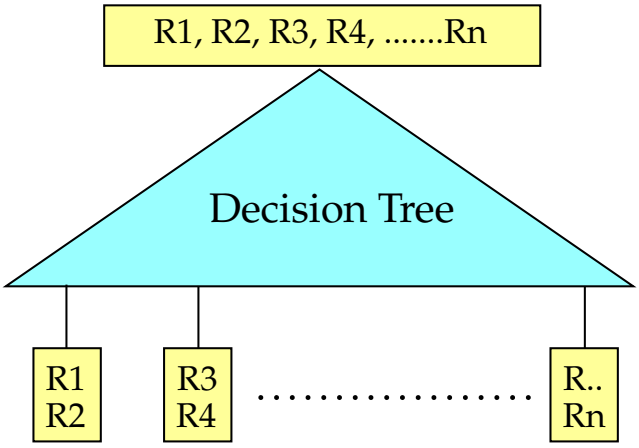
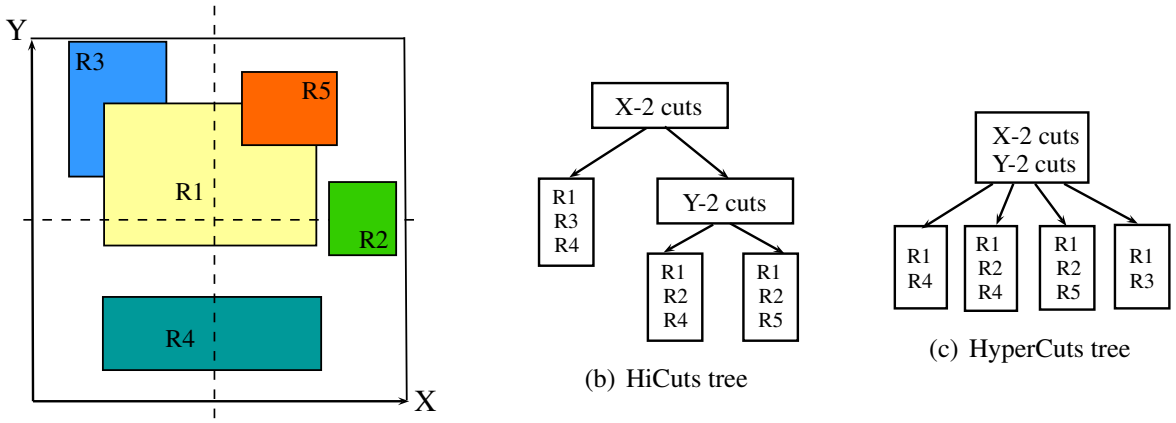


Figure 2.1: A general decision tree data structure used by packet classification system

Researchers proposed different types of software-based solutions for Packet classification in the recent years [6]. Among these solutions, decision tree-based algorithms [4, 12, 14] (e.g. Hicuts, HyperCuts) usually scale well. Decision tree based packet classification system constructs a decision tree from the rules in the rule-set where each leaf in the decision tree holds a small list of possible matching rules (Figure 2.1). Packet header fields are used to traverse the decision tree to reach a leaf node and a small amount of linear searching is used to find a highest-priority matching rule. In this chapter, we will briefly discuss about two most popular decision tree based solutions. We present novel techniques in Chapter 3 that can be applied to improve the look up performance of these decision tree based solutions.



(a) 2-dimensional rule-set

(b) HiCuts tree

(c) HyperCuts tree

Figure 2.2: Decision tree-based packet classification system

2.1 The HiCuts Algorithm

HiCuts builds a decision tree by partitioning the multi-dimensional rule space (e.g., source IP, destination IP, source port, destination port, protocol) into smaller segments with the goal of evenly distributing the rules into the tree's leaves. The root node of HiCuts covers the entire rule space. HiCuts cuts the space only along single dimension to create a set of equi-sized segments which separate the rules as evenly as possible. Each segment is represented by a child node. Figure 2.2(a) shows an example of a two-dimensional rule space. The figure shows five rules and the corresponding HiCuts decision tree (Figure 2.2(b)). The rules that span multiple subspaces are replicated in each of the corresponding children (e.g., R1, R2 and R4 are replicated Figure 2.2(b)). The cutting procedure is continued at each child node until the number of rules associated with the current node is less than a predetermined threshold called *binth* (e.g., 3 in Figure 2.2(b)). The nodes that contain less than *binth* number of rules are considered a leaf. Each leaf contains a set of pointers to its rules. The lookup algorithm is very simple. Packet header fields are used to traverse the decision tree until a leaf node is reached. The rules at the leaf node are then searched linearly to

determine the highest-priority matching rule.

For any given rule-set, there are possibly many ways to construct a decision tree, and HiCuts uses some heuristics based on structure present in the rule-set to guide the process of tree construction. The preprocessing algorithm in HiCuts uses a heuristic:

1. *to pick the number of interval cuts to make at each node*: HiCuts observes that a large number of cuts at a node will decrease the depth of the tree (which will accelerate look up time) at the expense of increasing storage due to increase in rule replication and also the number of children. To balance this trade-off, the heuristic attempts to maximize the number of cuts, and hence minimize the depth, while limiting the total number of rules at all the children of a node to be within a factor, called space_factor, of the number of rules at the node.

2. *to pick which dimension to cut along at each internal node*: HiCuts proposes various metrics that can be used to pick the dimension, including for example: the selected dimension should minimize the maximum number of rules per child resulting from the cut in an attempt to decrease the worst-case depth of the tree; or, select the dimension that has the largest number of distinct components (range) of rules in that dimension.

3. *to maximize the reuse of child nodes*: HiCuts observes that in real world rule-sets many child nodes have identical set of rules. The heuristic removes this redundancy by using a single child node for each distinct set of rules and have identical child nodes point to it.

4. *to eliminate redundancies in the tree*: The recursive cutting procedure used by HiCuts may introduce a scenario where a higher-priority rule completely overlaps a lower-priority rule within a nodes subspace. In this case, no packet would ever match the lower-priority rule. So, storage requirements can be improved by detecting and eliminating these redundant rules.

HiCuts has two main limitations. First, the tree depth depends on the distribution of the rules in the rule space. For some rule-sets, no matter how many cuts are going to be executed at a time, the HiCuts algorithm requires deeper levels in the decision tree. This limitation arises from the fact that HiCuts considers only one dimension to cut at a node. Second, the redundancy removal heuristic employed by HiCuts fail to remove redundancy completely. The heuristic can detect only the simplest form of full redundancy where some of the child nodes cover identical set of rules. However, it does not detect partial redundancy when some child nodes share many but not all the rules. For example, a heavily wildcarded rule often ends up in many leaves, increasing storage unnecessarily. This redundancy grows dramatically for the large size rule-sets.

2.2 The HyperCuts Algorithm

HyperCuts, the successor of HiCuts, addresses the limitations mentioned above. First, instead of cutting only one dimension (field) at a node, HyperCuts proposes to split the set of current rules at each node based on information from one or more dimension (fields) in the rule. Consequently, the data structure results in a fatter and shorter decision tree.

Second, HyperCuts partly addresses the redundancy where all the child nodes have associated a subset of rules that are identical. To remove this redundancy, HyperCuts employ a heuristic that simply moves all common rules in a subtree to a linear list at the root of the subtree. This heuristic is more flexible and efficient in redundancy removal because it is not limited by the fact that all the rules of a child node need to be common with another child node. Although, this optimization reduces replication but adds extra memory accesses to search the linear list held at each non-leaf nodes.

In HyperCuts, there are two types of nodes: internal nodes and leaf nodes. An internal node contains more than *binth* rules, where *binth* is a small constant that limits the amount of linear searching at leaves. Therefore, rules stored in the internal node have to be further partitioned to its child nodes. By contrast, each leaf node contains less than or equal to *binth* number of rules (e.g., 3 in Figure 2.2(c)) which are linearly traversed to find the matching rule.

The main strengths of HyperCuts are improvement in the tree depth and memory consumption. We choose HyperCuts over HiCuts in Chapter 3 because of its superior performance and scalability reported in the research community.

CHAPTER three

EFFICIENT MEMORY LAYOUT FOR PACKET CLASSIFICATION SYSTEM

In decision tree based packet classification system, packets are classified by searching in the tree data structure. Tree search presents significant challenges because it requires a number of unpredictable and irregular memory accesses. Since packet classification is per-packet operation and memory latency (caused by cache and TLB misses) is considerably high, any technique that can reduce cache and TLB misses can be useful in practice for improving lookup time in packet classification. In this chapter, we present an efficient memory layout for the tree data structure which ensures the movement of data optimally among the different levels of the memory hierarchy on general purpose processors. In particular, for a given node size, the number of accessed cache lines (and memory pages) is minimized by our proposed memory layout resulting in less number of cache and TLB misses. This reduction directly contributes in improving the look up performance. The decision tree laid out in the proposed layout can also exploit the strong computing power of multi-core architecture by leveraging data- and thread-level parallelism.

3.1 Introduction

The network devices, routers and firewalls, execute operations such as packet filtering, Quality-of-Service (QoS), traffic engineering for a specific subset of network packets using a technique called *Packet classification*. There are two major categories of solutions for performing packet classification: Hardware-based (using Ternary content addressable memory (TCAM)) and Software-based (using commodity processors and memory). TCAM-based solutions [23] can provide deterministic and high-speed lookup, but they are expensive and lack the adaptability to ever-changing network protocols. Also, TCAMs are not scalable with respect to power consumption compared to static

random access memories (SRAMs). For example, TCAMs consume 150 times more power per bit than that of SRAMs [6]. SRAM-based solutions offer the best flexibility and programmability. As a result, software based solutions (using SRAM) is still widely deployed on both edge routers and core routers.

Among the proposed software based solutions, decision-tree-based techniques received wider acceptance in the research community. HiCuts [12] and HyperCuts [4] are the two most popular decision-tree-based techniques. Packets are classified by searching in tree data structure. Tree search presents significant challenges because it requires a number of unpredictable and irregular memory accesses. Packet classification is per-packet operation and memory latency (caused by cache and TLB misses[1]) is considerably high. The growing trend of large rule-sets size coupled with the recent advancement in transmission link rates (Gbps) makes wire-speed classification a challenging task. Therefore, any technique that can reduce cache and TLB misses can be useful in practice for improving lookup time in packet classification. To reduce cache and TLB misses, *data locality* is one of the most important factors to be considered when designing the memory layout for the data structure. A fast processor's time is wasted when programs with poor data locality spend a significant amount of time waiting for data to be fetched from memory.

The search algorithm in HyperCuts tree involves comparing the search key to the key stored at a specific node at every level of the tree, and traversing a child node based on the comparison results. It is guaranteed that one of the child nodes in the next level will be traversed next. Hence, by carefully storing the child nodes close to its parent can improve the locality. Study [24] reported that cache misses can be reduced significantly by laying out data carefully.

[1] An auxiliary structure called the Translation Lookaside Buffer (TLB) which is used to perform a conversion from virtual to physical memory addresses prior to each memory access.

We, therefore, pursue to devise a memory layout for the HyperCuts tree which will reduce cache and TLB misses on general purpose processors. The idea is to try to keep the contemporaneously accessed elements of the data structure close to each other in the memory region. The spatial and temporal locality[2] is improved by this packing scheme. However, just naïvely co-locating child nodes to its parents may not help to achieve performance improvement because it does not exploit the underlying architecture features such as cache line size and page size efficiently. In this work, we propose techniques that take the architecture into considerations as well. Since HyperCuts tree does not change frequently, we believe that our proposed layout will reduce the cache and TLB misses on average due to its ability to utilize the memory hierarchy in an efficient way to achieve good data locality.

Packet classification system's performance can be further improved by utilizing the increased computing power of low-cost, highly parallel modern multi-core CPUs because its data parallel computing model is suitable for packet classification system.

The key **contribution** in this work is that we present a memory layout that rearranges tree nodes based on architecture features like page size, cache line size to reduce cache and TLB misses. We implement the proposed layout in multi-core architecture by employing data-level and thread-level parallelism during the classification phase, thereby exploiting the strong computing power in modern processors. We evaluate the proposed layout on two different state-of-the-art processors. Experimental results showed that our proposed memory layout provides significant performance improvements (40–55% faster) and achieves near-linear speedup (3.8× on quad cores) on multi-core architecture. We would like to emphasize that our proposed layout can be applied in general with any other decision tree-based packet classification system (such as Hicuts [12]) to improve the look up performance.

[2]The locality in memory access is often categorized into two different types, code reusing recently accessed locations (temporal) and code referencing data items that are close to recently accessed data items (spatial) [25].

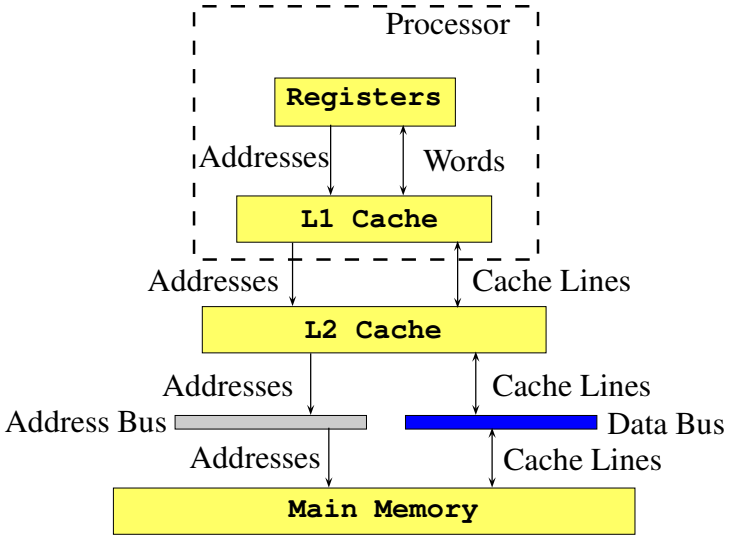


Figure 3.1: A typical memory hierarchy in modern computer architecture

3.2 Background

3.2.1 Overview of Memory Hierarchy and Cache

Modern computer hardware architecture contains a hierarchy of memory levels, with different memory sizes, block transfer sizes and access times and with each level acting as a cache for the next. Figure 3.1 gives a typical example of a memory hierarchy. The memory hierarchy consists of following components: registers, level-1 cache, level-2 cache, main memory, and disk. The level-1 cache is very small (typically 8 KB to 32 KB), very fast, very expensive, and very close to the CPU and its registers, usually on the CPU chip itself. The level-2 cache is larger (typically 128 KB to 2 MB), somewhat slower, and located either on the CPU chip or at least mounted on the same board. In some recent designs, even a third level of caching is used. Thus, instead of two levels in the memory hierarchy, namely CPU registers and main memory as traditionally considered in algorithm design, there is as many as five levels: registers, three levels of caches and main memory. For instance the Intel core i5 and AMD Opteron have 5 levels in its memory

hierarchy. The time for accessing a level in the memory hierarchy increases from one cycle for registers and level-1 cache to figures around 10, 100, and 100,000 cycles for level-2 cache, main memory, and disk, respectively. The evolution in CPU speed and memory access time indicates that the differences between them are likely to increase in the future [25]. As a consequence, the pattern of the memory access of data intensive main memory applications like packet classification will remain a key component in determining its performance in practice.

3.2.2 Multi-Core Architecture

In multi-core architecture, all processors are integrated on the same chip. Different cores execute different threads (multiple instructions) and operate on different parts of memory (multiple data). All cores have dedicated cache and they share the same address space as well. For example, Intel core i5 has 2 cores. Each of the cores has private L1 and L2 cache. Both the cores share L3 cache. High throughput data structures can benefit from the strong computing power provided by these multiple cores. However, the performance gains are limited by the extent to which the operations on the data structure can be parallelized to run on multiple cores concurrently. In this work, we employ thread-level parallelism and data parallelism to create parallel version of the HyperCuts.

3.3 Efficient Memory Layout for HyperCuts Tree

The goal of our proposed memory layout is twofold: First, laying out the HyperCuts tree in a way that ensures better data locality and second, to make the tree traversal operation compute-bound. The benefit of making compute-bound results in scaling with increasing number of cores (Section 3.5).

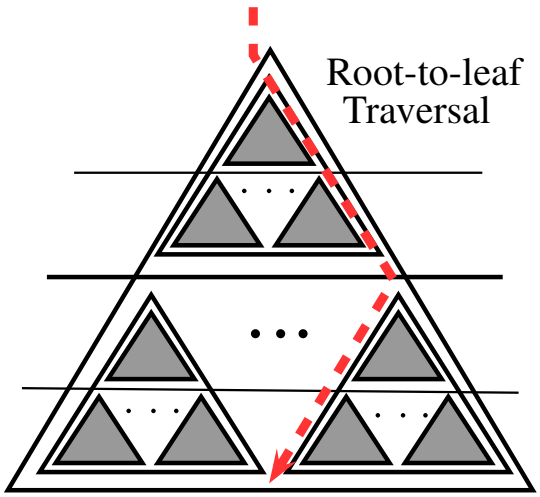


Figure 3.2: Root-to-leaf traversal in a tree laid out in hierarchical layout

The *van Emde Boas* layout (vEB), proposed by van Emde Boas et al. [26], aims to speed up the tree based search by ensuring that a group of data which are likely to be accessed contemporaneously during an execution are placed contiguously in the memory region. However, we observe that just naïvely applying vEB layout is not effective when the tree data structure is larger than the last level cache (LLC). This is because in order to improve cache locality, we need to ensure that temporally coherent data are stored close to each other i.e. we need to store all the children nodes close to their parents in a tree data structure. However, doing so for all depths results in a breadth-first storage. This increases the storage distance between any node and its children at deeper depths. When the tree is large, traversing the bottom part of the tree results in memory accesses that are separated by increasing distances. Since memory is laid out as pages on modern architecture, each access results in accessing nodes stored in different pages of memory incurring TLB misses at each level and search becomes latency bound. In addition, if the working set is too big to fit in caches, we need to ensure that a cache line which has been transferred from the memory is fully utilized before being evicted out of caches. Unfortunately, there is no dedicated

mechanism in vEB layout to utilize cache-line efficiently.

To accomplish our first goal, we layout the data structure in a way similar to vEB layout. However, to accomplish our second goal we propose a scheme called "hierarchical blocking" that consists of two key techniques called "cache line blocking" and "page blocking". The main purpose of the hierarchical blocking scheme is to reduce cache and TLB misses. A cache line with a typical size of 64 bytes can contain multiple nodes in it (e.g. 4 nodes of 16 bytes each). Our cache-line blocking technique ensures that the data in a single cache line is utilized more efficiently. This efficiency is achieved by performing a rearrangement of tree nodes within each page so that the subsequent nodes to be used also stays within the same cache line. Our page blocking technique addresses the problem associated with traversing the bottom part of the large tree mentioned above. This technique rearranges the tree nodes so that the contemporaneously accessed tree nodes also stays within the same page, thereby ensuring good cache locality. We expect that the root-to-leaf tree traversal (Figure 3.2) of the HyperCuts tree laid out according to our proposed approach will incur less number of cache and TLB misses on average.

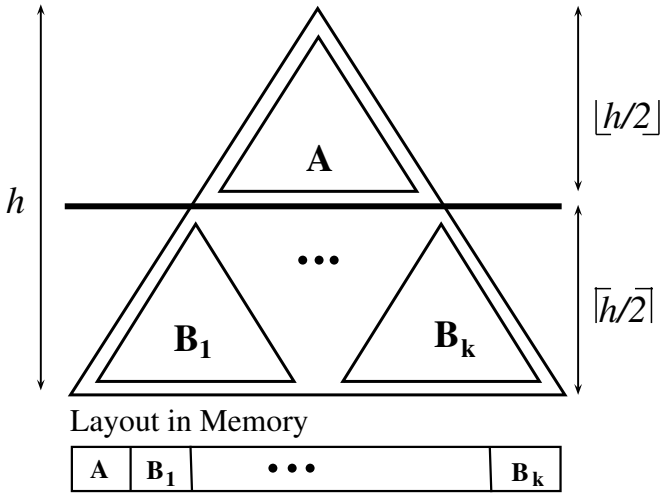


Figure 3.3: Recursive hierarchical layout of a tree in memory

3.3.1 Hierarchical Layout

In order to introduce the layout, we will use a simple binary tree example. By layout here, we mean the mapping of the nodes of a binary tree to the indices of an array where the nodes are actually stored. The nodes should be stored in the bottom array in the order shown (Figure 3.3) for achieving better locality and hence searches to be fast.

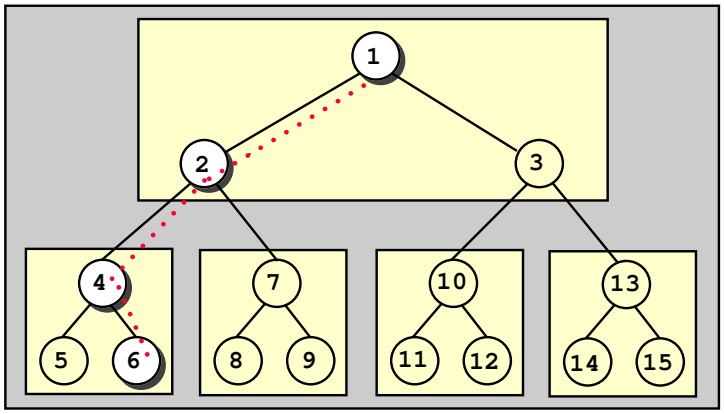


Figure 3.4: Binary tree laid out in hierarchical layout

Given a complete binary tree, we describe a mapping from the nodes of the tree to positions of an array in memory. Suppose the tree has N items and has height $h = \log N + 1$. Split the tree in the middle, at height $h/2$. This breaks the tree into a top recursive subtree of height $\lfloor h/2 \rfloor$ and several bottom subtrees B_1, B_2, ..., B_k of height $\lceil h/2 \rceil$. There are $\sqrt{N}$ bottom recursive subtrees, each of size $\sqrt{N}$. The top subtree occupies the top part in the array of allocated nodes, and then the B_i's are laid out. Every subtree is recursively laid out (Figure 3.3). The order of the recursive subtrees is not important; what is important is that each recursive subtree is laid out in a single segment of memory, and that these segments are stored together without gaps. This strategy provides some benefits on spatial locality by grouping the children and their parents in nearby locations. To handle trees whose height h is not a power of two the split rounds so that the bottom

recursive subtrees have heights that are powers of two, specifically $\lceil h/2 \rceil$. This process leaves the top recursive subtree with a height of $h - \lceil h/2 \rceil$, which may not be a power of two. In this case, we apply the same rounding procedure to split it.

The binary tree in Figure 3.4 is shown with the nodes labeled with their positions according to the layout mentioned above. In this example, only two levels of recursion are needed. If we suppose that every page stores three nodes, then the highlighted path from node 1 to 6 visits only two pages incurring less cache and TLB misses on average.

3.3.2 HyperCuts Tree in Hierarchical Layout

Table 3.1: Example rule-set; Source and Destination port ranges cover 4-bit port numbers

RuleID	*Src Port*	*Dst Port*
1	1:3	12:14
2	10:11	6:13
3	8:9	0:15
4	0:15	2:6
5	1:6	1:10
6	10:11	1:2

In this section, we will show a demonstration on constructing a HyperCuts tree from the rule-set in Table 3.1 and laying out the HyperCuts tree according to the hierarchical memory layout (Section 3.3.1).

The geometric view of the rule-set in 2-dimension is presented in Figure 3.5. On the plane are six rectangles, each representing a rule. HyperCuts uses simultaneously cutting multiple dimensions with the goal of evenly distributing the rules into the trees leaves. In this demonstration, the dimensions to cut and number of cuts along each dimension are picked up following heuristics described in [4]. The cutting procedure is continued until the number of rules in a node is less than

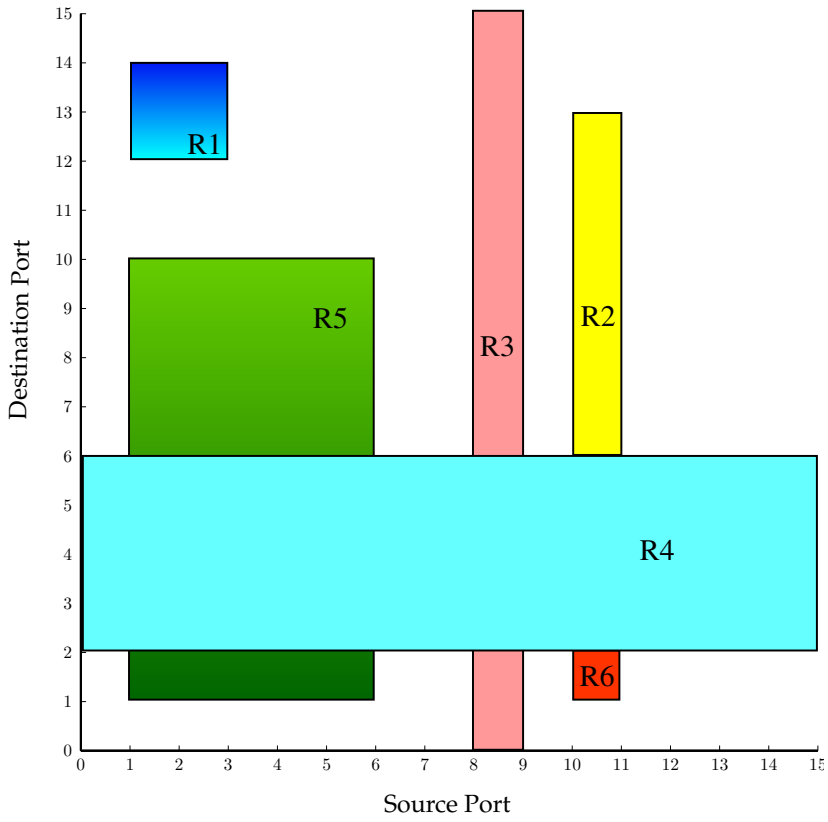


Figure 3.5: Geometric representation of the example rule-set (Table 3.1) before the cutting sequence

binth. For this example, $binth = 2$. The cutting sequence given by the heuristic is as follows: at the first step, we give 4 cuts along the x-axis and 2 cuts along the y-axis. It will generate 8 sub-regions in total. Each sub-region is a node in the tree (the link to empty sub-region is not shown in Figure 3.7). At this point, one sub-region overlaps more than 2 rectangles i.e. one node in the tree contains more rules than the predefined threshold *binth*. So the cutting procedure continues. Next, we cut this sub-region along the x-axis. Now, we have another sub-region that overlaps 3 rectangles.

So, the cutting procedure continues. Next, we cut this sub-region along the y-axis. Now each sub-region overlaps $= 2$ rectangles. We can stop cutting the space further because no node in

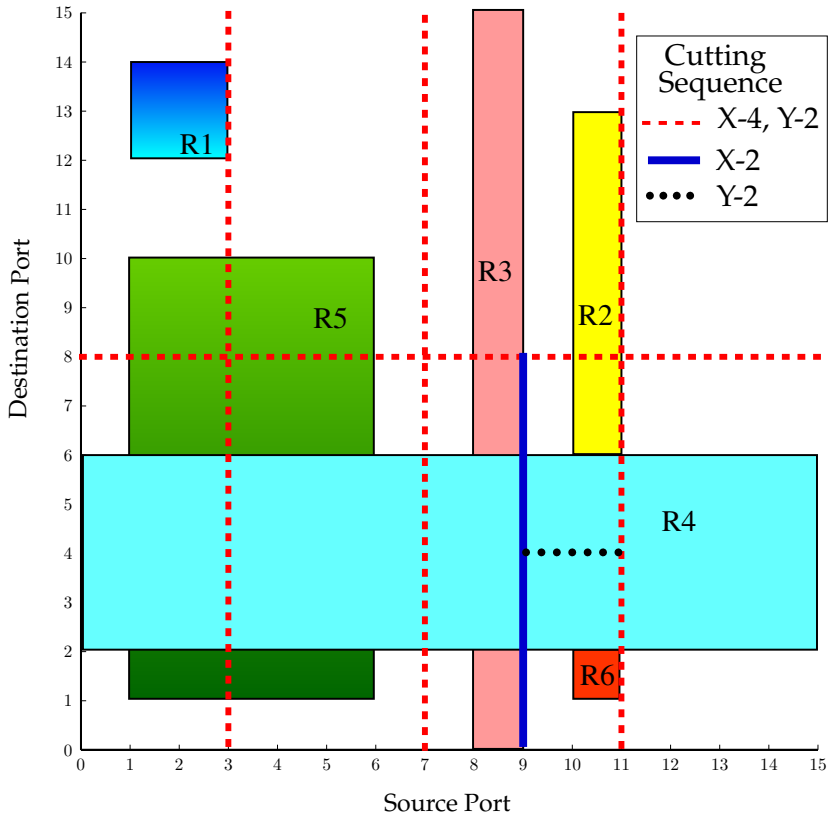


Figure 3.6: Geometric representation of the example rule-set (Table 3.1) after the cutting sequence

the tree contains more than *binth* number of rules. The effect of cutting is displayed in geometric representation in Figure 3.6 and the resulting decision tree from this cutting sequence is shown in Figure 3.7. Since the tree height[3] is 4, we divide in the middle (according to hierarchical layout) and we lay out the top sub-tree in contiguous memory. Bottom subtree is also laid out in contiguous memory (recursively) using the same principle (Figure 3.8).

Suppose, a packet just arrived which will match ruleID 4. If every page stores the shaded region in the tree, then search for the matching rule will visit only two pages, thereby reducing the possibility of cache and TLB misses on average.

[3]We assume that height of root node is 1.

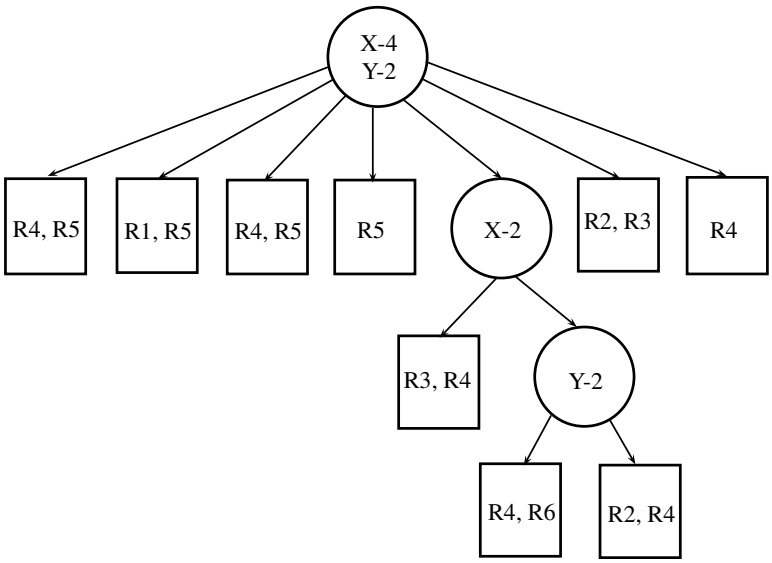


Figure 3.7: HyperCuts tree for the rule-set in Table 3.1

3.3.3 Hierarchical Blocking

The proposed hierarchical blocking scheme contains two key techniques: page blocking and cache line blocking. The goal of page blocking technique (solid triangle in Figure 3.9) is to cluster the nodes into groups of N nodes where N is the number of nodes that fit entirely in a memory page (typical size of 4 KB). Furthermore, the nodes are rearranged within every page so that the children and their parents are stored in nearby locations - thereby fully exploiting a cache-line (dotted triangle in Figure 3.9). In our implementation, the node size is 16 byte. As a result, we can fit 4 nodes in a cache line (typically 64 bytes or longer) and $125K$ nodes in a 2 MB memory page.

During the initialization phase of page blocking technique, all the nodes in the tree are marked as unassigned node to any cluster. A record of the list of unassigned nodes is maintained throughout the process. The threshold Δ is set to N. All the nodes in the tree are assigned depth values (using a depth-first traversal starting with the root node). The clustering procedure begins by selecting one of the unassigned nodes at the lowest depth. For example, we start clustering with

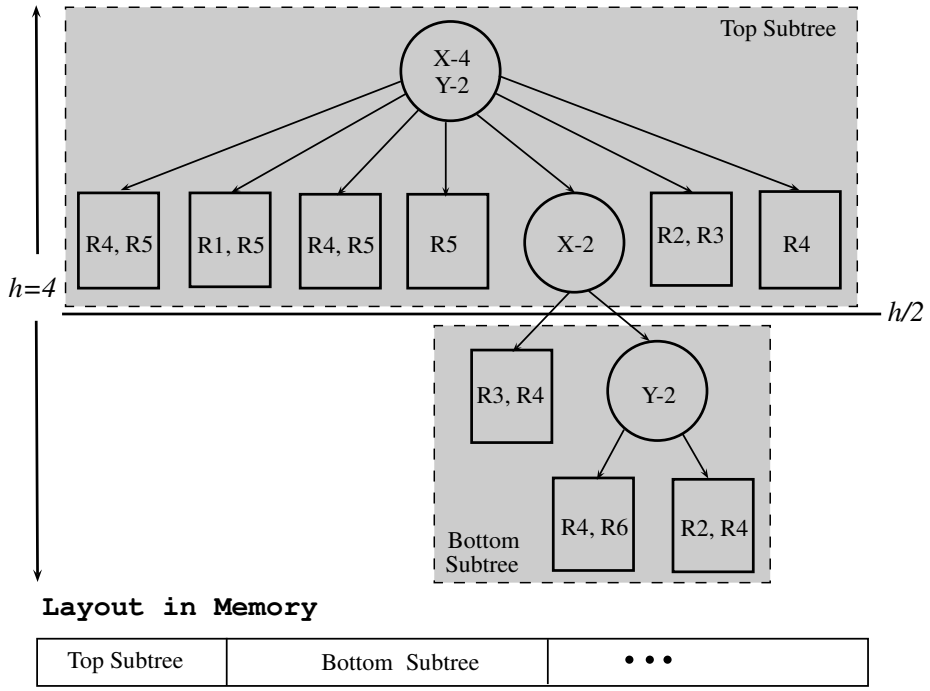


Figure 3.8: HyperCuts tree laid out in hierarchical layout

the root node of the tree and consider the sub-tree with $N-1$ nodes. The selected node and its unassigned children are then included in the cluster. This process is carried out till the threshold Δ is reached. The clustering for the next page is started by either continuing with the children of the node just being selected or starting with a new unassigned node. The clustering process is complete when all the tree nodes have been assigned to any of the clusters.

The cache line blocking is performed within each cluster that has just been created using the page blocking technique. During the initialization phase, all the nodes within the cluster being considered are marked as node unassigned to an index. A record of the list of nodes which have not been assigned an index is maintained throughout the process. We begin with the node at the lowest depth and consider its children in the cluster. In case, the number of unassigned children is greater than 4 (the maximum number of node within a cache-line), we assign the children contiguously.

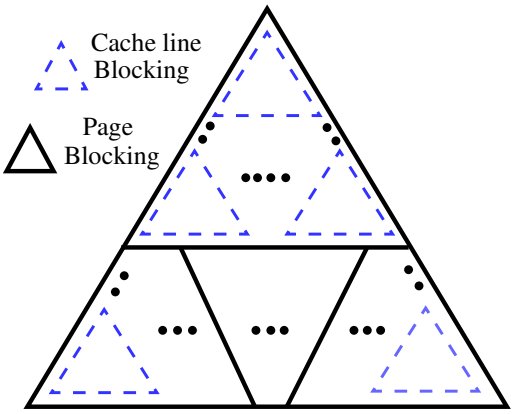


Figure 3.9: HyperCuts tree blocked in two-level hierarchy - 1st level Page blocking and 2nd level Cache line blocking

We continue with the remaining children and assign them in a similar fashion. If the number of unassigned children is less than 4, we fill up the cache-line partially, and continue with the process by picking up the unassigned node at the lowest depth. The process is complete when all the tree nodes within the cluster have been assigned an index.

Our hierarchical blocking scheme aims to reduce the average number of cache and TLB misses for any distribution of the incoming packets. In practice, the tree traversal spends most of its time in the first few levels of the tree, which are clustered together by our scheme. Hence, at an average we have very few cache and TLB misses. Our blocking scheme helps to obtain significant improvement in look up performance for tree size larger than LLC (Section 3.5).

3.3.4 Analysis on Memory Access Pattern

We now discuss the memory access pattern with our proposed hierarchical blocking scheme in action. Let us define some notations that we will use in this section. Let, d_{tree} is the depth of the tree, d_{page} is the tree depth of page blocking, d_{cl} is the tree depth of cache line blocking, l_{page} is the latency incurred when there is a TLB miss, l_{cl} is the latency incurred during the cache line fetch.

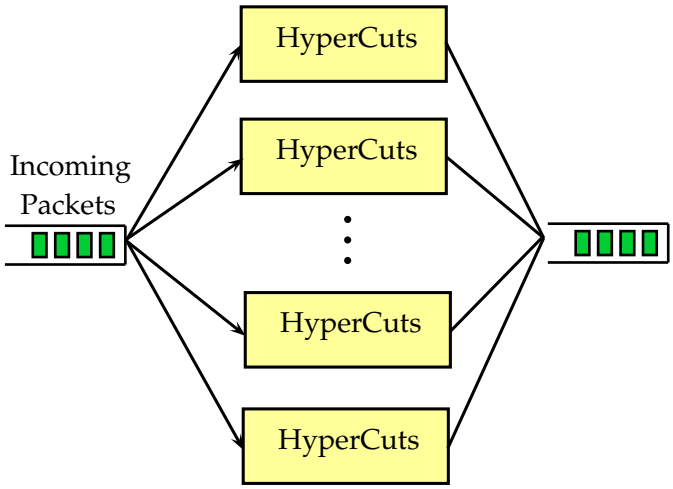


Figure 3.10: Data parallel HyperCuts. Each core executes the complete algorithm.

At the beginning of the experiment, we have cold cache and TLB. Consequently, the comparison to the root accesses a memory page and incurs a TLB miss. This will cost a latency of l_{page} cycles. Then the appropriate cache line is brought from the main memory, incurring a further latency of l_{cl} cycles. Now, we access the necessary nodes within this cache line cluster (i.e. within the next d_{cl} levels). The subsequent access incurs cache miss which costs a latency of l_{cl} cycles. On average, we say that within the first page, $\lceil d_{page}/d_{cl} \rceil$ cache lines will be accessed. So, any memory page would incur a total latency of $(l_{page} + \lceil d_{page}/d_{cl})\rceil l_{cl})$ cycles. Traversing to the bottom part of the tree would access $\lceil d_{tree}/d_{page} \rceil$ pages, for an average incurred latency of $\lceil d_{tree}/d_{page} \rceil (l_{page} + \lceil d_{page}/d_{cl})\rceil l_{cl})$ cycles. To take into account of different levels in memory hierarchy, suppose, the last level cache contain d_{llc} out of d_{tree} levels of the tree. Although the TLB size in modern processor is not infinite, we can reasonably assume that for a random packet distribution the entry for the top page would be in the page table during the lookup process. Therefore, the lookup process would incur a latency of $(1 - d_{llc}/d_{tree})(\lceil d_{tree}/d_{page} \rceil \lceil d_{page}/d_{cl} \rceil l_{cl}) + l_{page}(\lceil d_{tree}/d_{page} \rceil - 1)$ cycles on average.

3.4 HyperCuts on Multi-Core Architecture

We considered and implemented data parallel approach (Figure 3.10) to create parallel HyperCuts algorithm on multi-core architecture. In data parallel approach, the classification algorithm is fully replicated onto each core, and packet headers are distributed to an arbitrary core for parallel classification. The scalability of this straightforward approach is reasonably good because inter-thread communication is not strictly needed when classifying two separate packets. Construction of data structures are done before the replication, and a single copy of these and other read-only data structures is shared among all instances of the algorithm.

Figure 3.10 illustrates the data parallel approach for the HyperCuts algorithm. We use the producer consumer model in our implementation. One thread is capturing the incoming network packet (produce data) and other threads are classifying the packets (consume data). Packet headers are placed in a queue shared between the producer and the consumer(s). We use mutual exclusion lock to properly guard the shared queue against the concurrent accesses and the race conditions. On each core, the classification cycle consists of following three stages:

1. busy-waiting to acquire the mutual exclusion lock
2. retrieving a header from the queue
3. perform the classification by finding the matching rule

The data parallel approach has two following merits:

1. *simplicity*: applications parallelized using data parallel approach requires only the inclusion of a mutual exclusion lock and some logic at the entry point of the algorithm to synchronize the queue. In addition, no extra effort is required to extract parallelism out of the algorithm.

2. *minimal dependency*: dependencies between cores are minimal in data parallel approach. The classification on different cores can run independently without regard to the speed or duty cycle of other cores. In addition, cores do not need to execute in tight synchrony.

3.4.1 Mutual Exclusion by Atomic Operation

During the experiment, we find out that the use of a single lock guarding the shared queue limits the scalability with increasing the number of threads. Consequently we do not observe a linear speed-up as the number of competing threads increases. Our investigation reveals that the contention for acquiring the shared queue's lock is a dominant factor. Our finding is also consistent with the prior study [27] where Mellor-Crummy and Scott find that this construction limits scalability for even small numbers of processors.

This bottleneck is mitigated somewhat in CMP architecture by exploiting the implicit degree of freedom in the choice of synchronization primitive. Study [28–30] found that the use of atomic variable to implement locking are the least expensive among other synchronization primitives such as condition variables, pthread mutexes using busy-wait loops. In order to reduce the lock acquisition time for synchronization, we, therefore, opt in to use atomic variable in lieu of condition variables or library-provided locking. The main characteristic of atomic variables is that while it is being accessed by one thread, no other thread can interrupt it. This is why they are called atomic variables. In practice, atomic variables are the best solution to arbitrate the access to a simple variable (queue's counter) shared among two or more competing threads.

There are twelve functions in *gcc* atomic builtins [31], which guarantee atomic memory access. These functions do atomic add, substitution, and logical atomic or, and, xor and nand. There are two functions for each operation. One that returns value of the variable before changing it and another that returns value of the variable after changing it. The built-in function

"__sync_fetch_and_add(*type* *ptr, *type* value, ...)" available in *gcc* provides atomic add operation. This function adds the value indicated in the second argument to a memory location specified in the first argument, and returns the value that had previously been in memory. The steps of the operation are as follows:

```
{ tmp = *ptr;
*ptr += value;
 return tmp; }
```

The entire set of steps are done as an atomic operation, which means that only one thread can perform this update at a time. This call is much faster than a mutex. We use this function to arbitrate access to the shared queue in our implementation and this simple optimization helps to observe near-linear speed-up with increasing number of competing threads during the experiment (Section 3.5).

How Atomic Variables Work

Modern processor architectures has instructions that allow one to lock Front Serial Bus (FSB), while doing some memory access. The core uses this bus to communicate with memory, i.e. access to the memory by any other core, and threads running on that core can be prevented by locking FSB. This is exactly what is needed to implement atomic variables [32].

3.5 Evaluation of Hierarchical Layout

We evaluate the effectiveness of our proposed memory layout on several architectures. The different processor architecture varies mainly in terms of number of cores and L1, L2, and L3 cache size. We implement our hierarchical blocking scheme on an Intel Core i5 and a quad-core AMD

Table 3.2: Configuration details of test beds

Name	*Test bed I*	*Test bed II*
Processor Name	AMD Opteron 2379 HE	Intel Core i5
Core Count	4	2
Core Speed (MHz)	2400 (per Core)	1330 (per Core)
L1 Cache Size (KB)	128 (per Core)	32 (per Core)
L2 Cache Size (KB)	512 (per Core)	256 (per Core)
L3 Cache Size (MB)	6 (shared by 4 cores)	3 (shared by 2 cores)

Opteron processor. The detail configurations of the machines are given in Table 3.2. We use the C language and POSIX pthread-library [33] because of their simplicity and portability. We conduct experiments on Linux OS. Each experiment is repeated a number of times and results are averaged.

We generate synthetic rule-sets with characteristics representative of real-world rule-sets using ClassBench [34]. All the rules in the rule-sets are 5 dimensional tuples composed of source and destination IP addresses, source and destination port numbers and protocol type. The size of the five rule-sets ranges from 1K to 10K. The synthetic packet headers used in our experiments were generated using the ClassBench trace generator. In order to test the impact of our proposed optimizations on look up performance, we conduct the experiments with two extreme cases: small trees (tree size smaller than L2 cache) and large trees (tree size larger than LLC). The large tree is generated from the 10K rule-set and the small tree is generated from the 1K rule-set. We do not present the performance of intermediate tree sizes because it falls in between the above two cases.

We compare the normalized search time, measured in cycles per lookup on Intel Core i5 CPU. The tree node size is 16 byte in our implementation. The cache line size is 64 bytes and the page size used in our study is 2 MB, although smaller traditional pages of size 4 KB are available. We first present the naïve lookup measurement when no optimization techniques are used. Then, we present the improvement obtained from each optimization such as page blocking, cache line

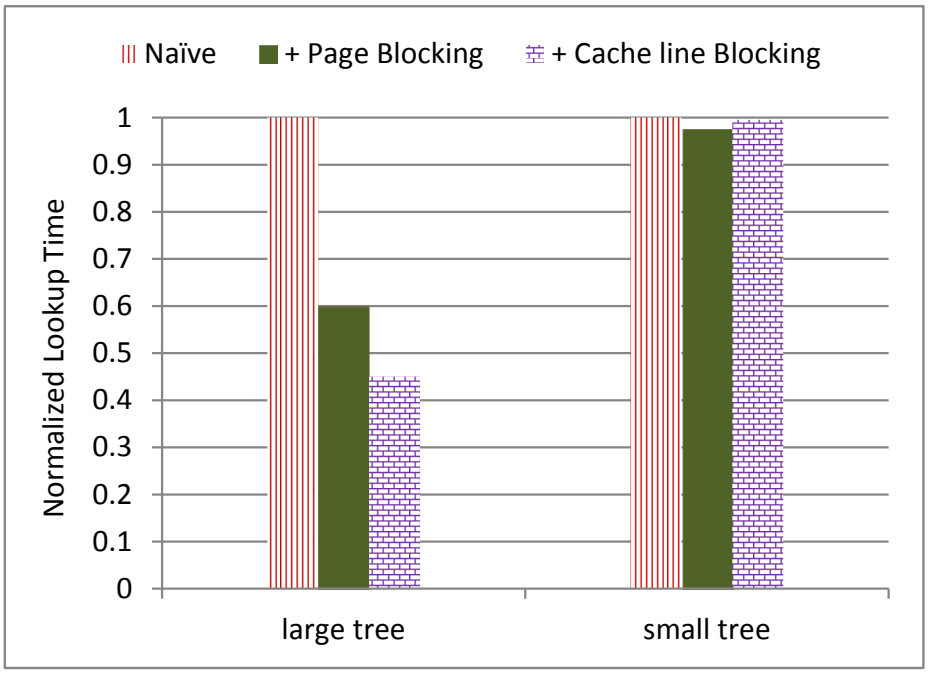


Figure 3.11: Normalized lookup time with various optimization techniques (lower is better)

blocking.

According to the Figure 3.11, the impact of each optimization is more visible for large size trees than small size trees because as the tree size grows, lookup process suffers from cache and TLB misses and large trees are more latency bound than the small trees. The page blocking technique obtains 40% faster lookup performance. Augmenting cache line blocking with page blocking results in more faster lookup performance (55%). The reduction in cache and TLB misses on average while traversing the bottom part of the large tree significantly contributes to the improvement in lookup time. However, for small trees our techniques do not bring any improvement because for small trees Intel Core i5 CPU is able to store the entire tree within L2 cache and consequently there are no TLB and cache misses. The results reported here are based on a single-thread execution.

In Figure 3.12 and Figure 3.13, the x-axis represents the number of nodes in log scale and the y-axis represents the average number of cycles. The impact of our proposed optimization tech-

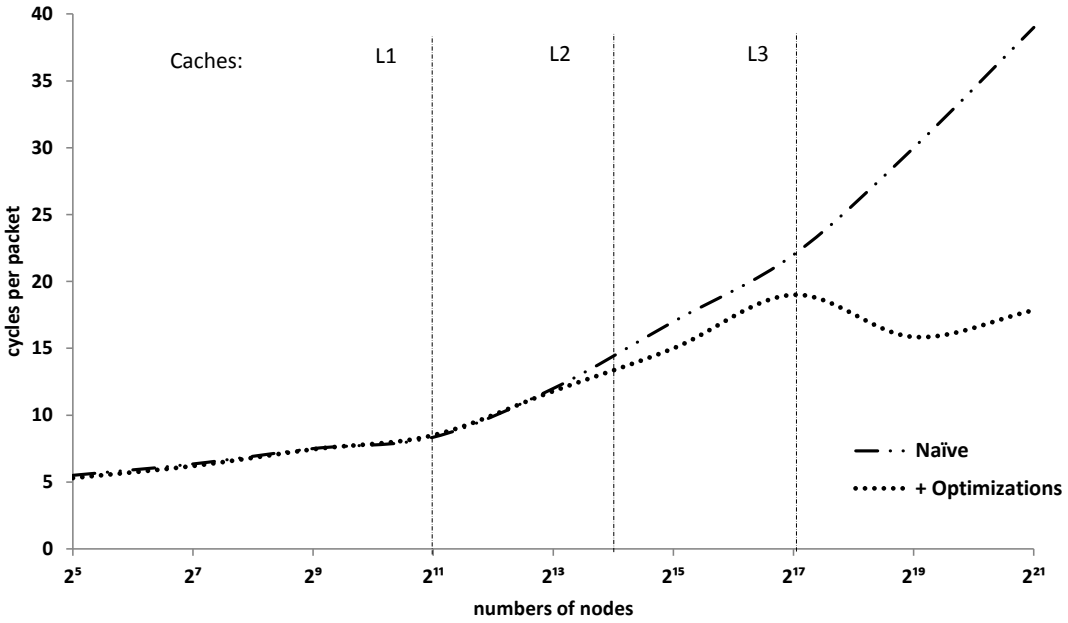


Figure 3.12: Look up performance on Intel Core i5 processor (lower is better)

niques are reported on both Intel Core i5 and AMD Opteron platform. We observe that the impact of our proposed optimization techniques on both platforms are more noticeable when the tree size starts to grow beyond LLC. This result validates our claim that by applying the hierarchical blocking scheme, lookup process can suffer from reduced amount of cache and TLB misses for tree sizes larger than the LLC.

In our final experiment, we seek to find out whether the HyperCuts tree laid out in our proposed memory layout coupled with the optimization techniques can leverage the strong computing power of modern multi-core general purpose processors. Note that our intention is not to show that the optimization techniques contribute on the speed up on multi-core architecture rather we want to confirm that the expected speed up due to multi-threading is not inhibited by the introduction of our optimization techniques. Five synthetic rule-sets have been used in this experiment. The rule-set sizes ranges from 1K to 10K. We start the experiment by executing HyperCuts (HyperCuts-1)

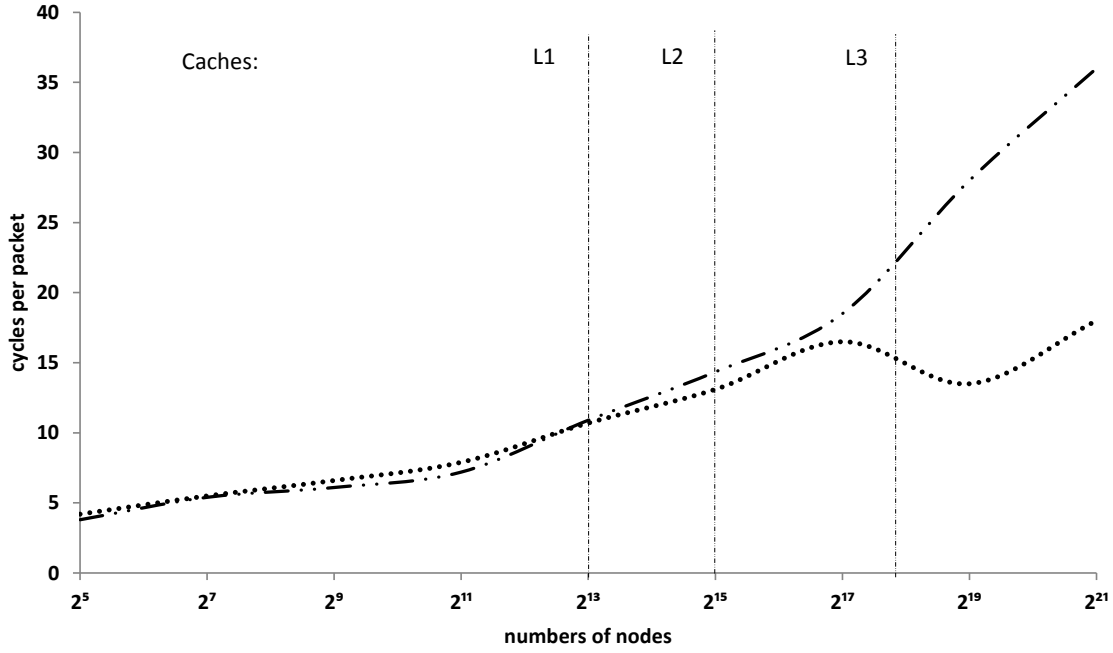


Figure 3.13: Look up performance on AMD Opteron processor (lower is better)

on a single core. No parallelism is exploited in HyperCuts-1. We then gradually start utilizing more cores by leveraging data parallel scheme. In the HyperCuts-4 multi-threaded version, each of the 4 cores of AMD Opteron performs packet classification concurrently on different set of input packets. Note that construction of the HyperCuts tree and laying out in proposed memory layout is performed prior to multi threading, and a single copy of the tree structure is shared among all the threads. In order to avoid context switching and process migration costs [35], each thread is hard-affinitized to the core using `pthread_setaffinity_np` function. The throughput reported in Figure 3.14 is the *maximum throughput achievable* by both HyperCuts (single threaded and multi-threaded) without packet loss. The maximum throughput achievable of HyperCuts is the maximum incoming packet rate in which HyperCuts does not suffer from any packet losses. In Figure 3.14, the *x*-axis represents the five rule-sets, and the *y*-axis is the packet classification throughput. The throughput gain ranges from 2.5 to 3.8 on quad-core AMD Opteron machine.

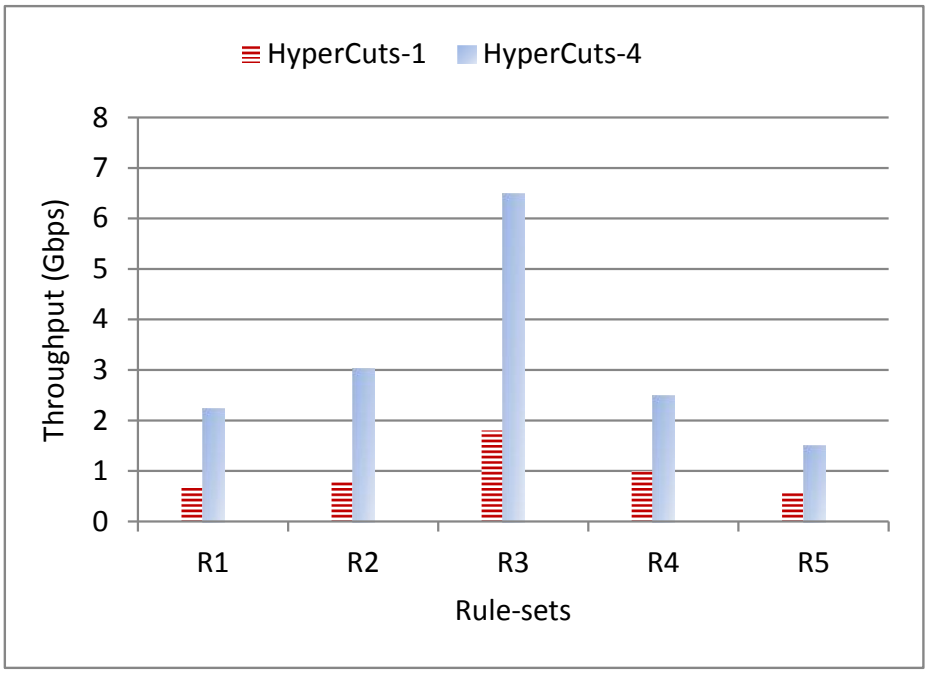


Figure 3.14: Throughput gain of HyperCuts tree (both optimization techniques applied) on quad-core AMD Opteron

3.6 Conclusion

We show that faster lookup time can be achieved by rearranging the nodes of the HyperCuts tree in an order more appropriate for the memory hierarchy of current modern general purpose processors. Packet classification is per-packet operation and its performance is dominated by memory references because it requires a number of unpredictable and irregular memory accesses during the search for matching rule for each packet. Poor memory layout can easily generate weak data locality and poor performance. In this work, we use an efficient hierarchical memory layout for decision tree based packet classification system to improve the data locality. The goal of this layout is to carefully placing the tree nodes close to each other in memory that are accessed closely in time, thereby reducing the cache and TLB misses on average. The decision tree laid out in

the proposed layout can also exploit the computing power of multiple cores on general purpose processors by leveraging data- and thread-level parallelism. We evaluate the proposed layout on two different state-of-the-art processors. Experimental results show that our proposed memory layout provides significant improvements (40–55% faster) in lookup process on both platforms and achieves near-linear speedup (3.8$\times$ on quad cores) on multi-core architecture. Finally, we would like to emphasize that our proposed layout can be applied in general with any other decision tree-based packet classification system (such as Hicuts [12]) to improve the look up performance.

CHAPTER four

TRAFFIC ADAPTIVE PACKET CLASSIFICATION SYSTEM

Packet classification is primarily used by network devices, such as routers and firewalls, to do additional processing for a specific subset of packets. Such additional processing includes packet filtering, quality of service (QoS), and differentiated services (DiffServ). Most of the existing packet classification algorithms reported in the literature exploits the characteristics of filtering or classifier rules in optimizing their techniques. We pursue to find out whether that packet classifier's average performance can be improved by exploiting the locality in the traffic pattern. In this chapter, we present `npf`, a fast, lightweight, traffic-adaptive packet classification system. Our lightweight traffic-aware packet classifier reorganizes its internal data structure (rule tree) based on the traffic pattern to reduce the search time for the most frequently visited rules in the rule-set. Unlike existing traffic-adaptive packet classifier requiring a separate, offline reorganization phase, our approach performs reorganization online with little overhead, resulting in improved look up time per packet on average.

4.1 Introduction

Today's Internet traffic consists of different types of traffic, such as delay sensitive, malicious or benign. Thus, every Internet router or firewall today needs to treat these traffic mix differently. Packet classification is the mechanism that enables the routers and firewalls to classify the traffic and process them accordingly. Packet classification becomes a core mechanism used by a variety of Internet applications including Security, traffic monitoring, Quality of Service (QoS). For example, an Intrusion Detection System (IDS) classifies the packet either as benign or malicious based on the policy in the rule-set. The rule-set consists of multiple rules that characterize the pattern of

the malicious activity. Individual entries for classifying a packet are called rules. For instance, a typical firewall rule specifies that packets from which subnet should be blocked in order to protect network against malicious attacks. In this context, the packet classification problem is to determine the first matching rule for each incoming packet.

The rapid improvement in link speed at the core and edge of the Internet and the growing size of rule-set [4] pose challenges on the existing traditional packet classifiers. Therefore, the design of an efficient fast packet classification algorithm still remains an important open problem. Packet classification is per-packet operation and the search speed is a major performance factor. It requires a number of high latency memory accesses (slower than the link speed). Therefore, designing a Packet classifier that spends minimal time per packet (i.e. incurs small number of memory accesses) is inherently a challenging problem.

Previous works in packet classification resulted in two threads of solutions: *hardware-based* (TCAM-based) and *software-based* packet classification (optimization of underlying data structures). Although the hardware-based packet classification shows promises in performance optimization, it has some potential issues such as slow growth rate of TCAM chip size, high power consumption and high cost. In addition, the need for rules with range specifications to be translated into several CAM entries is expensive for very large rule-sets. Software-based packet classification which has been studied extensively in the research literature [5] either exploit the characteristics of rule-set in their optimization techniques or try to optimize their technique for the worst case scenario by minimizing the depth of the search tree (Hi-Cuts [12] and HyperCuts [4]).

The prior observations [2, 10, 12, 20–22] that a relatively small subset of rules are visited more frequently in the rule-set show promise to another direction that packet classifier's average performance can be improved by exploiting the locality in the traffic pattern. Therefore, we seek to exploit these observations to produce a traffic-aware decision tree based packet classification

system `npf` [36] that is able to exploit the locality in the traffic pattern to achieve a near-optimal searching time on average. Recently, Adel [1] et al. proposed a traffic-adaptive packet classifier. However, their approach performs a separate, offline reorganization of the underlying data structure at regular interval. This may introduces mediocre performance if Internet traffic pattern changes before the next reorganization interval. In order to address this issue, we undertook the investigation of finding the feasibility of performing online dynamic adaptation to the incoming traffic pattern. In particular, our interest focuses on analyzing the tradeoff between the performance gain and processing overhead due to dynamic adaptation and performing a comparative study between online and offline adaptation approach. In the experiment, we use `npf` that performs dynamic reorganization (online) of the underlying data structure along the changes in the traffic pattern.

4.2 Importance of Traffic Awareness in Packet Classifier

Most of the existing packet classification approaches exploit the characteristics of rule-set. For example, Florin et al. [16] made rearrangement of rules in the rule-set with the assumption that in real rule-set the rule overlap is rare. This rearrangement of rules reduces the number of memory accesses which improves the packet classification performance overall. Another underlying crucial assumption of their work is that the number of rules that match a packet is small. Our study in Section 5.2 observes that a small subset of all rules in the rule-set are visited (matched) more frequently by the the majority of the inbound or outbound packet and this traffic skewness property is likely to stay for time intervals that are sufficient to make such skewness important to consider in designing algorithm for packet classification [1]. This observations along with other findings reported in the literature motivate us to design and develop a traffic-aware packet classification system. A detail analysis on the motivation of exploiting traffic pattern is presented in Section 5.2.

The main contribution of this work is the design, implementation and the analysis of per-

formance gain and overhead of a packet classification algorithm (`npf`) that dynamically adapts its internal tree structure to the properties of Internet flows and packet headers in reducing the classification time. In `npf`, a tree structure is used to store the rules and it is dynamically reorganized to exploit the locality in the incoming traffic. The goal of the adaptation to the traffic pattern is to ensure that most frequently visited rules will be found early during the search process, thereby incurring less memory accesses. Packet headers are used to traverse the tree and it accesses memory as the look up process traverses down the tree. In order to consume less number of memory accesses, we promote the tree node associated with the most visited rules closer to the root of the tree. Traffic awareness does not cause any oscillation in the underlying data structure reorganization because we observe from a large number of Internet and private traces that most of the packets match only a few rules and this characteristics of Internet traffic is unlikely to change over a short period of time. The online reorganization introduces overhead. But we believe that the amortized overhead is much smaller than the overall performance gain.

4.3 Related Work

Most simple packet classification technique uses linear search through the rule-set to find a matching rule. However, for a large rule-set, this memory efficient technique consumes unacceptably long search time. Specialized data structures, and heuristics have been proposed in order to speed up the software-based packet classification. The significant contributions to the advancement of packet classification research made by the previous works aim to improve the worst-case matching performance. Hence, they do not exploit the the pattern to improve the average packet matching time. The techniques that exploit the traffic pattern for optimizing the performance of packet classification was discussed in [1, 37–39].

Lukas et al. [37] proposed a technique to make search structures adapt to traffic dynamics

by inserting a shortcut path from the root of the search tree to the frequently visited nodes which reduces the number of memory accesses. This method performs a control loop, where periodically the statistics about the popularity of the node are read and the shortcuts for the next time interval are placed into the tree. However, they showed that their method only works for Denial-of-Service (DoS) attack traffic.

The authors [38] introduced the idea of statistical data structures in optimizing packet filtering. In this work, depth-constrained alphabetic trees are used to reduce lookup time of destination IP addresses of packets against entries in the routing table. They showed that the average case lookup time (finding matching rule) can be significantly reduced by using statistical data structure. However, their approach was limited to only one field (routing prefix) with arbitrary statistics because the focus of the paper is on routing lookup. In addition, the paper does not provide any details on traffic statistics collection and dynamic update of the statistical tree.

The use of statistical trees for optimizing packet classification is also found in [21]. The authors proposed a technique that used whole rules without exploiting traffic patterns over separate fields.

Researchers [1, 39] utilized traffic pattern over separate fields to obtain adaptive methods that can accommodate varying traffic statistics for achieving a near-optimal average matching time if the traffic statistics get stable over time. They build Huffman trees using traffic space segments where all segments are disjoint subsets of the traffic space such that each subset contains flows that have some common characteristics. Their approach adapts to the traffic pattern in regular intervals (offline).

Our approach is similar to the existing traffic adaptive solutions in the way that we also take traffic pattern into consideration to improve the average packet matching time. However, the novelty of our approach lies in: *(i)* adaptation is done as traffic comes in (instead of regular

intervals) to match the most-recent traffic characteristics *(ii)* adaptation is done "online" and *(iii)* our technique provides much finer granularity than whole rules with the added capability to digest multiple fields.

4.4 Proposed Approach

Traditional packet classifier classifies packets based on a rule-set, being unaware of the traffic pattern. However, `npf` goes one step further. In addition of performing its main duty—"Packet classification", `npf` intelligently reorganizes its internal data structure by exploiting locality in the incoming traffic in order to achive a near-optimal searching time.

`npf` is a decision tree based algorithm. It constructs a special data structure called "Interval based Rule Tree (IRT)" in order to efficiently store the rules in the rule-set.

The dynamic adaptation to the traffic pattern in `npf` is done by using a technique called "tree rotation". Applying this technique, the nodes associated with the most frequently visited rules are stored at the shortest path in the search tree. This results in a significant matching reduction for the most popular traffic. It is highly likely that the next search for the matching node will be found quickly by consuming less number of memory accesses because the nodes associated with the most frequently visited rules is expected to be found near the root of the "`IRT`" (Section 4.6). In best possible case, our `npf` will find the matching rule for the most popular traffic in at most one memory access.

We perform two kinds of rotation: *left rotation* and *right rotation* based on the location of the node to be rotated in the tree. The rotation operation preserves the binary-search-tree property. When we do a *left rotation* on a node x, we assume that its right child y is not `NULL`. x may be any node in the tree whose right child is not `NULL`. The *left rotation* works around the link from x to y. It makes y the new root of the subtree, with x as y's left child and y's left child as x's right child. On

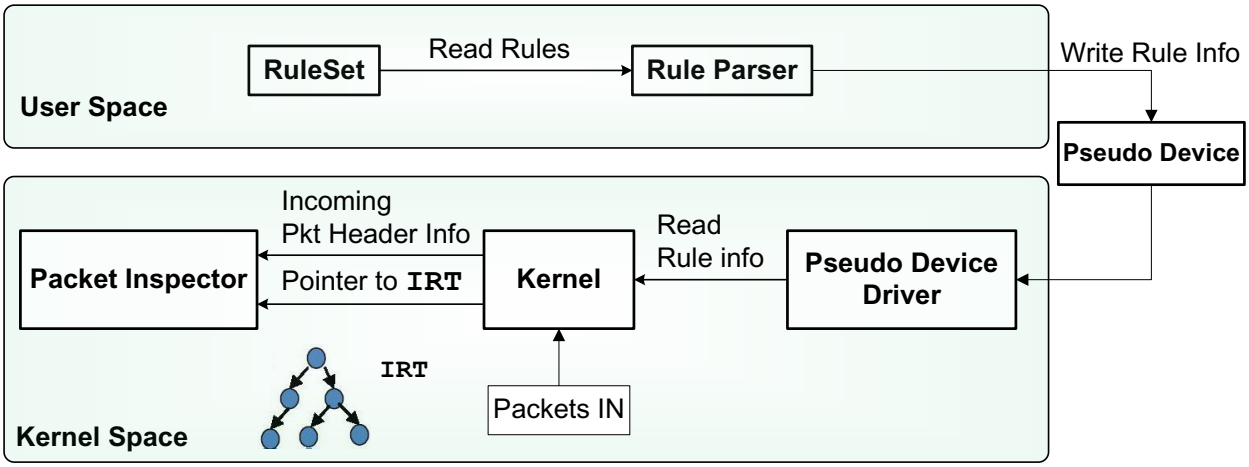


Figure 4.1: High level view of `npf` design

the other hand when we do *right rotation* on a node x we assume that its left child y is not `NULL`. It makes y the new root of the subtree, with x as y's right child and y's right child as x's left child [40]. Both *left rotation* and *right rotation* run in $O(1)$ time. These rotations have two important effects: they move the node being rotated upward in the tree, and they also shorten the path to any nodes along the path to the rotated node. This latter effect means that rotation operations tend to make the tree more balanced. The other advantage of tree rotation is that rule dependencies/ordering on the `IRT` hierarchy are preserved because each node contains the list of matching ruleIDs.

The high level design of `npf` is shown in Fig. 4.1. Our proposed design mainly consists of 2 phases.

- *Phase I: Preprocessing Phase* constructs the decision tree `IRT` and list of ruleIDs for wild-card rules based on the rules in the classifier in the user space for each of the five dimensions (source/destination IP, source/destination Port, and protocol). These information are down-loaded to the kernel and then `IRT` is rebuilt in kernel space.

- *Phase II: Search Phase* queries multiple `IRT`s to identify the matching rule for the incom-

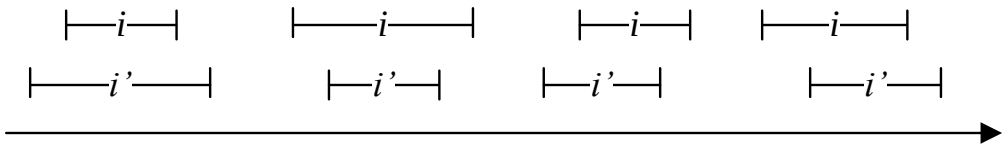


Figure 4.2: The interval trichotomy for two closed intervals i and i'. If i and i' overlap, there are four situations: in each $low[i] \leq high[i']$ and $low[i'] \leq high[i]$

ing packet. Kernel modules update the popularity counter of the matching node and performs dynamic reorganization of the tree based on the traffic pattern. The main objective of our approach was exploiting traffic pattern to minimize the maximum number of operations (memory accesses) to be performed during the search phase.

Each time a packet arrives, the `IRT` for each dimension (field) is traversed based on the information in the packet header to find a matching node in the respective `IRT`. We obtain for each field a set of candidate rules that contain the corresponding field value. The rule(s) that matches the packet are obtained by getting the intersection between these sets of rules. If the intersection contains more than one rule, the rule with highest order (priority) is selected. If no rules are common, then the default action is returned. An important advantage of this concept is that the field searches can be performed more efficiently because of the relatively small search keys and by exploiting specific field characteristics. Another advantage is that the memory accesses related to different field searches are independent and can be performed in parallel or in a pipelined fashion (Chapter 5), which reduces the latency and allows a better utilization of the available memory bandwidth. We describe the design and working procedure of `npf` in detail in the following sections.

4.4.1 Phase I: Building `IRT` from Scratch

The algorithm starts with a set of N rules where each of the rules containing K dimension (field). In this work, we consider $K = 5$ dimensions i.e. source/destination IP, source/destination Port and protocol in the rule-set. However, `npf` can be extended for higher number of dimensions if needed. Since packet matching is performed on multiple fields, multiple `IRT` are constructed. The tree structure is named as "Interval based Rule Tree" because everything (IP address and port number) is considered in terms of interval or range. For example, 16 bit port field ranges from 0 to $2^{16} - 1$. Thus the port field has an interval of $[0, 65535]$. We also convert the IP prefix into appropriate interval. We can represent an interval $[t_1, t_2]$ as an object i, with fields $low[i] = t_1$ (the low endpoint) and $high[i] = t_2$ (the high end point). We say that the interval i and i' overlap if $i \cap i' \neq \emptyset$, that is, if $low[i] \leq high[i']$ and $low[i'] \leq high[i]$. Any two intervals i and i' satisfy the *interval trichotomy*; that is, exactly one of the following three properties holds [40]:

1. i and i' overlap
2. i is to the left of i' (i.e., $high[i] < low[i']$)
3. i is to the right of i' (i.e., $high[i'] < low[i]$)

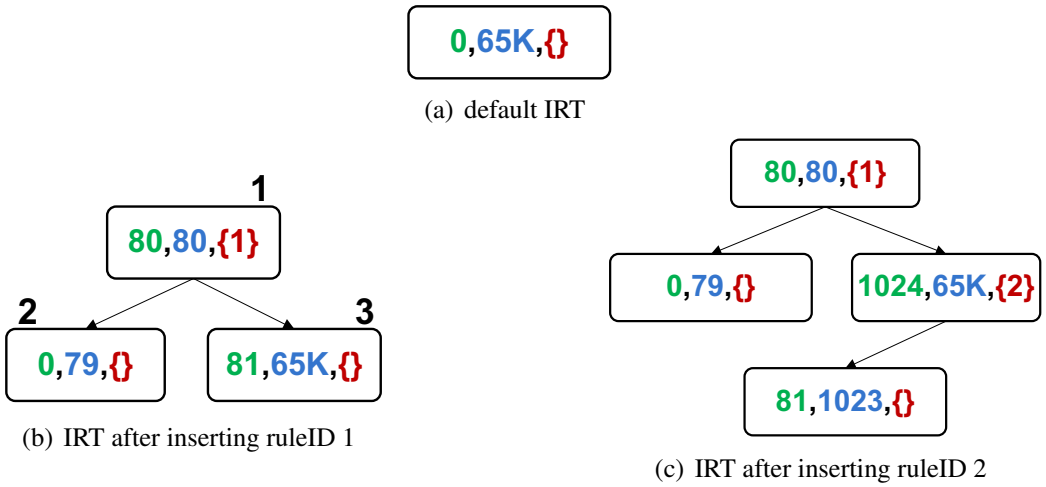
Fig. 4.2 shows the all possible interval trichotomy for two closed intervals i and i'. If the interval specified by two rules overlap (partially or fully), the higher priority rule will overwrite the lower priority rules in the `IRT`. The ordering of the rule in the rule-set typically is the priority indicator. The rules in the top order are usually considered higher priority rules than the rules in the bottom order.

Each node x in the decision tree has the following properties: an interval [`begin range, end range`], `popularity` of the interval and {`list of ruleIDs`} matching this interval.

`begin range` is the starting point of the interval or range, `end range` is the end point of the interval. For example, 16 bit port field that has interval $[0, 65535]$, `begin range` = 0 and `end range` = 65535. `popularity` defines the frequency of the interval [`begin range`, `end range`] seen in the incoming traffic. {`list of ruleIDs`} stores the list of rules that contains the interval [`begin range`, `end range`] in rule-set. The `IRT` is created based on the *binary-search-tree* property. Let x be a node in `IRT`. If y is a node in the left subtree of x, then $end\_range[y] < begin\_range[x]$. If y is a node in the right subtree of x, then $end\_range[x] < begin\_range[y]$. An in-order tree walk of `IRT` lists the intervals in sorted order by start point of the interval or `begin range`. The worst case search time in binary tree is $\lg n$ where n is the number of elements in the tree. However, the binary tree does not exploit the the skewed distribution property of the traffic as described in Section 5.2. To exploit this property, we employ techniques in `IRT` in order to minimize the average matching time. `IRT` basically is reorganized so that the nodes associated with the most frequently visited rules are stored at the shortest path in the search tree. This way, field values that commonly exist in the traffic will incur less number of memory accesses in comparison to less frequent values, resulting in a significant reduction in the matching time of most popular flows, reducing the overall average classification time of all flows. Although the `IRT` based tree matching may not be in favor of less-frequently matched traffic, it still improves the overall average classification by significantly reducing matching of most popular packets. The more the skewness in the traffic distribution over field values, the more the gain in the classification performance. Even when the traffic distribution is uniform (in the worst case scenario), `npf` cannot do worse than the binary search as a lower bound. We can argue that the worst case scenario is unlikely to occur for all the header fields at the same time for long time. The analysis presented in Section 5.2 corroborates this.

Table 4.1: An example of packet classifier

ruleID	SrcIP	DestIP	SrcPort	DestPort	Protocol	Action
1	*	*	[600,610]	80	TCP	Accept
2	*	*	[1000,1024]	[1024,65535]	TCP	Reject



(a) default IRT

(b) IRT after inserting ruleID 1

(c) IRT after inserting ruleID 2

Figure 4.3: Construction of Interval based Rule Tree `IRT`

In order to store nodes associated with the most frequently visited rules at the shortest path in the search tree, we employ the second constraint *max-heap property* in `IRT` . The *max-heap property* mandates that for any non-root node the popularity must be less than to the popularity of its parent. Thus, the node with most popular interval (i.e. most frequently visited rule) are stored at root node and its left and right subtrees are formed in the same manner from the subsequences of the sorted order to the left and right of that node.

We initialize the `popularity` of each node associated with an interval based on the number of its descendants. It helps us to preserve the *max-heap property* of `IRT` . Then, when we find a matching node during search operation the popularity counter is incremented. In order to find whether the max-heap property is violated, we compare the popularity of a node to its parent's

popularity. On violation of max-heap property, we use the tree rotation technique to regain the max-heap property. Tree rotation promotes the higher popular nodes towards the root of the tree and demotes the less popular nodes towards the leaf of the tree. Consequently, frequently matched nodes (i.e. most popular rules) would be more likely to be closer to the root of the tree, causing searches for them to be faster [41]. All the nodes in `IRT` encompass the entire header field space, for example for Port field, it encompasses from 0 to 65535.

Fig. 4.3 shows the steps involved in `IRT` construction for "Destination Port" dimension (field) from rule-set (Table 4.1). The node contain `begin range`, `end range` and {`list of ruleIDs`}. For the sake of simplicity, we omit the popularity counter from the node in the tree.

Fig. 4.3(a) is the default node in the `IRT` at the beginning of `IRT` construction. The default node initially contains `begin range` $= Min(Interval) = 0$, `end range` $= Max(Interval) =$ 65535 and `list of ruleIDs` $= \{NULL\}$. Rule 1 has 80 in destination port field. Since it is a non-range value, so we have: `begin range` $= 80$, `end range` $= 80$ and `list of ruleIDs` $= \{1\}$. When we traverse the current `IRT` (Fig. 4.3(a)), we find out that interval $[80,80]$ overlaps with the interval $[0,65535]$ (interval trichotomy 1) [36]. Interval $[80,80]$ will overwrite the current node and thus the intermediate `IRT` (Fig. 4.3(b)) is constructed. We provide a marking system to the node in `IRT` (Fig. 4.3(b)) for ease of reference. This is not part of our design. The destination port of rule 2 is $[1024,65535]$ which means that rule 2 has an interval $[1024,65535]$. We traverse the current `IRT` (Fig. 4.3(b)) to find out its appropriate location to insert it into the tree. At node 1, we find out that interval $[80,80]$ is to the left of the interval $[1024,65535]$ (interval trichotomy 2) [36]. So we make a right branch traversal. At node 3, we find out that interval $[81,65535]$ overlaps with the interval $[1024,65535]$. Now interval $[1024,65535]$ overwrites the node 3 and `IRT` (Fig. 4.3(c)) is constructed. Without loss of generality, using this procedure the `IRT` can be constructed for other dimensions as well regardless of the number of rules in the

rule-set.

Algorithm 4.4.1: LOOKUP($root, packet\_header\_field$)

$t \leftarrow root$

$key \leftarrow packet\_header\_field$

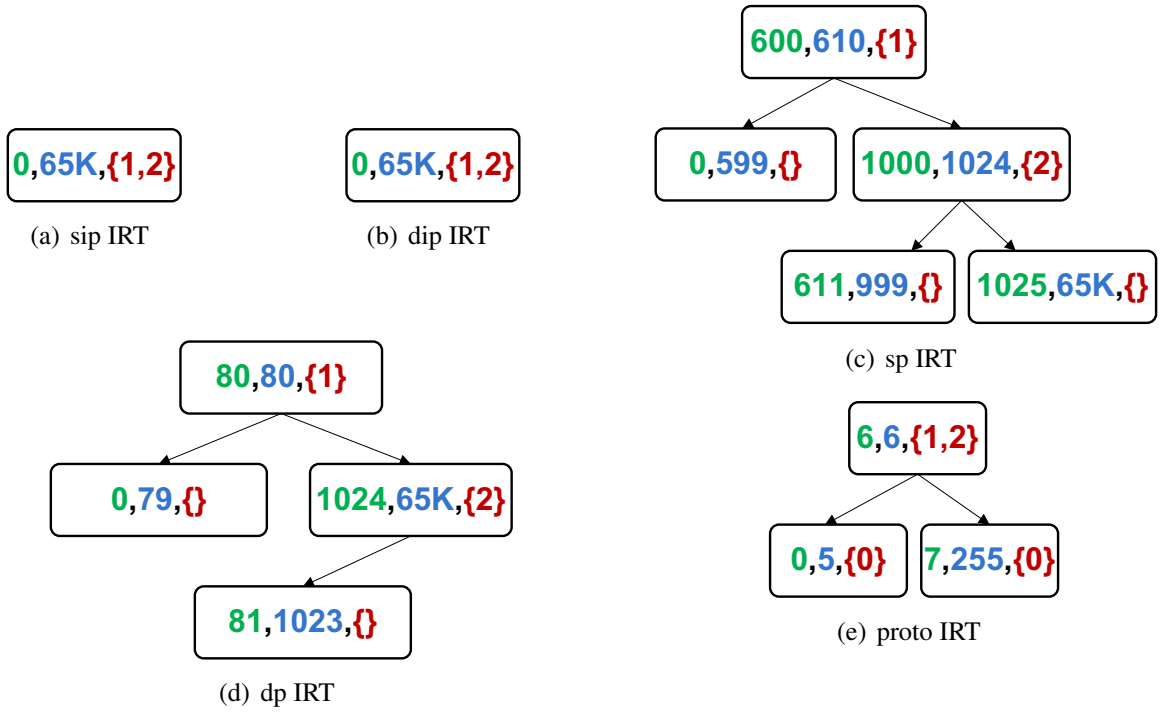
while $(t \neq NULL)$

do
- **if** $key < t \rightarrow begin\_range$
  - **then**
    - **comment:** Traverse the left subtree
    - $t \leftarrow t \rightarrow left$
  - **else if** $key > t \rightarrow end\_range$
    - **then**
      - **comment:** Traverse the right subtree
      - $t \leftarrow t \rightarrow right$
    - **else**
      - **comment:** Matching node found
      - $(t \rightarrow popularity) = (t \rightarrow popularity) + 1$
      - **if** $(t \rightarrow popularity) == (t \rightarrow parent \rightarrow popularity) + UPDATE\_THRESHOLD$
        - **then**
          - **comment:** Perform either left or right tree rotation
      - **comment:** Terminate the look up
      - **return** (t)

4.4.2 Phase II: IRT Search Algorithm

When a packet arrives we extract the value of the $K = 5$ dimensions (fields) from the packet header (SIP, DIP, SP, DP, and Protocol). During search process, the IRT for each dimension is traversed, and a matching node in each IRT is identified. For example, we traverse SIP IRT to find out a matching node that contains the interval such that $\texttt{begin\_range} \leq \texttt{SIP} \leq \texttt{end\_range}$. We do this search operation for identifying a matching node in IRT for other dimensions as well. The pseudocode to look up packet header field in IRT is presented in Algorithm 4.4.1. Traversing the trees yield 5 matching nodes. We collect the list of rules from the matching nodes. As a result, we get 5 different sets of ruleID list. We also have the list of wild card rules for each dimension from *Phase I*. The list of rules are unified with their wild card rules as follows: $U_{\text{SIP}} = \{\text{SIP}\} \cup \{\text{SIP}^*\}$, $U_{\text{DIP}} = \{\text{DIP}\} \cup \{\text{DIP}^*\}$, $\{U_{\text{SP}}\} = \{\text{SP}\} \cup \{\text{SP}^*\}$, $U_{\text{DP}} = \{\text{DP}\} \cup \{\text{DP}^*\}$ and $U_{\text{PROTO}} = \{\text{PROTO}\} \cup \{\text{PROTO}^*\}$. All of these unified lists of rules are intersected in the following order of $U_{\text{SIP}} \cap U_{\text{DIP}}$, $(U_{\text{SIP}} \cap U_{\text{DIP}}) \cap U_{\text{SP}}$, $((U_{\text{SIP}} \cap U_{\text{DIP}}) \cap U_{\text{SP}}) \cap U_{\text{DP}}$ and $(((U_{\text{SIP}} \cap U_{\text{DIP}}) \cap U_{\text{SP}}) \cap U_{\text{DP}}) \cap U_{\text{PROTO}}$. Here, $\{\text{SIP}\}$ contains the list of rules of the matching node and $\{\text{SIP}^*\}$ contains list of wild card rules for the source IP dimension. The intersection operation finally selects either a list of common ruleIDs or $\emptyset$. If the result of $\cap$ is not empty, ruleAction associated with the smallest common ruleID determines the fate of the packet— accept or reject, otherwise the fate of the packet is determined by the default ruleAction set by the Network Administrator. Note that npf focuses on traditional rule-set based on IP 5-tuples. Hence, in this work, we describe npf's packet classification process for $K = 5$ dimensions. However, npf's packet classification ability is not limited by the number of dimensions.

In order to make npf traffic pattern adaptive, statistics are collected using a simple calculation (e.g., counter increments). More specifically, the popularity counter for the matching node (i.e. a specific interval) is incremented during tree search operation. We compare the popularity



(a) sip IRT (b) dip IRT (c) sp IRT (d) dp IRT (e) proto IRT

Figure 4.4: Interval based Rule Tree `IRT` for five different header fields. "SIP" means Source IP, "DIP" means Destination IP, "SP" means Source Port, "DP" means Destination Port and "proto" means protocol. The legend of the node is as follows: `begin of interval`, `end of interval`, {`list of rule IDs`}

of the matching node with the popularity of its parent node. If the *max heap property* is violated (i.e. popularity(child node) = popularity(parent node) + Update_Threshold) then we apply the tree rotation to promote the higher popular nodes towards the root of the tree. As a result, searching time for the most frequently visited rules is reduced further which contributes in significant improvement of average case performance of packet classification. In addition, as soon as our algorithm finds that the result of $\cap$ operation between two dimension is $\emptyset$, it can bypasses the search operation and $\cap$ operation for other dimensions and classify the packet faster (because of saving memory accesses) without compromising the accuracy because $\cap = \emptyset$ implies that the fate of the packet will be determined by the default `ruleAction`. This inherent advantage present in our

approach contributes further in achieving superior average performance.

4.4.3 `npf`'s Packet Classification Demonstration

In this section, we demonstrate the packet classification procedure of `npf` with the help of a simple example. Table 4.1 is a very simple rule-set that contain only 2 rules. For the sake of simplicity we keep the number of rules in this rule-set small. Any incoming packet will match rule 1 if it is a TCP packet with source port within the range $[600, 610]$ and destination port 80, regardless of its source and destination IP addresses. Rule 2 will be matched when the packet is coming from source port within the range $[1000, 1024]$ and destined for any port within the range $[1024, 65535]$, regardless of its source and destination IP addresses. The five `IRT`'s (Fig. 4.4) for five different dimensions are constructed according to the procedure mentioned in Section 4.4.1.

Table 4.2: Packet header information

source IP	69.166.49.179
destination IP	69.166.48.145
source port	600
destination port	80
protocol	TCP

Now suppose a packet arrives at NIC whose header information is listed in Table 4.2. At first the source IP `IRT` (Fig. 4.4(a)) is traversed and the list of rules $\{\text{SIP}\} = \{1, 2\}$ is obtained from the matching node. We do the same for destination IP `IRT` and obtain $\{\text{DIP}\} = \{1, 2\}$. The list of wild card rules for source and destination IP dimension are $\{\text{SIP}^*\} = \{1, 2\}$ and $\{\text{DIP}^*\} = \{1, 2\}$, respectively. If $\{U_{\text{SIP}}\} \cap \{U_{\text{DIP}}\} \neq \emptyset$ the source Port `IRT` is traversed in order to find a matching node for source port 600. The search finds $\{\text{SP}\} = \{1\}$ because 600 lies between the interval $[600, 610]$ at the root node of (Fig. 4.4(c)). Here, $\{\text{SP}^*\} = \emptyset$. Again, if $(U_{\text{SIP}} \cap U_{\text{DIP}}) \cap U_{\text{SP}} \neq \emptyset$ the destination Port `IRT` is searched for a matching node for destination port 80 and we obtain

$\{DP\}=\{1\}$. Here, $\{DP^*\} = \emptyset$. Since $((U_{SIP} \cap U_{DIP}) \cap U_{SP}) \cap U_{DP} \neq \emptyset$, finally, we search proto `IRT` to find a matching node for TCP protocol and the search finds $\{PROTO\} = \{1,2\}$. Here, $\{PROTO^*\} = \emptyset$. The result of $(((U_{SIP} \cap U_{DIP}) \cap U_{SP}) \cap U_{DP}) \cap U_{PROTO} = \{1\}$. Hence, the fate of this packet will be decided by the action associated with rule 1.

Let us consider that another packet arrives with the same header information except the source port equals to 1024. Performing search in source IP and destination IP `IRT` will yield the following ruleID list: $\{SIP\} = \{1,2\}$ and $\{DIP\} = \{1,2\}$. Since $U_{SIP} \cap U_{DIP} \neq \emptyset$, the source port `IRT` is searched which finds $\{SP\} = \{2\}$. The intersection operation $(U_{SIP} \cap U_{DIP}) \cap U_{SP} \neq \emptyset$. So we search destination port `IRT` and we obtain $\{DP\} = \{1\}$. However, $(U_{SIP} \cap U_{DIP}) \cap U_{SP} \cap U_{DP} = \emptyset$. Therefore, the fate of this packet will be determined by the default `ruleAction` right away instead of searching other dimension `IRT` (in this case, proto `IRT`), resulting in faster classification. This bypass helps us improving the average performance of our packet classifier.

4.5 Attacks and Defense

`npf`, a traffic-adaptive packet classifier, exploits the pattern of the traffic to achieve a better performance. The crucial assumption to make `npf` efficient is that the traffic pattern would match only a small subset of rules in the rule-set and this pattern would show steady behavior for sufficient long interval. Unfortunately, the attacker can exploit this fact to force `npf` to have lower performance than expected. In this section, we present possible attack scenarios that are tailored specifically for statistical packet classification techniques. The main objective of this section is to give some ideas how different attacks can be planned against `npf` and what countermeasures could be taken to prevent the attack.

4.5.1 Attack on the adaptation method by injecting short term variation in the traffic pattern

The underlying reason of `npf`'s superior average performance is that the search time for the most frequently visited rules are reduced by dynamically promoting the tree node associated with those rules closer to the root of the tree. Hence, an attacker can lower the performance of `npf` by creating artificial short term variations in traffic pattern by injecting traffic flow with the least expected properties (e.g. invalid protocol, least used port numbers) which forces `npf` to traverse long path to find the matching node. As a result, these packets will take longer time to be classified by `npf` than the average packet. Consequently, the average throughput will decrease since one bad packet will increase the search time for all successive distinct packets.

4.5.2 Defenses

We describe the possible defense mechanisms of `npf` following the idea presented in [1].

`IRT` reconstruction

The performance of `npf` is measured by the number of comparisons (memory accesses) performed to classify a packet. Hence, we can detect the performance degradation of `npf` by observing the average number of comparisons (i.e. average path length) performed for every packet. Once the average cost per packet is exceeded above the predefined threshold, the search structure (`IRT`) is reconstructed. However, under an attack, the `IRT` may need to be rebuilt so frequently that it may overload the processing capacity of the system by rebuilding operations. One way to prevent this from happening, we can redefine the threshold which will force the `IRT` rebuild operation to occur not so frequently.

Dynamic Sampling rate

We perform sampling on the incoming traffic because it is hard to process each and every packet due to high speed link. We can mask out burst attack by carefully choosing sampling rate. By lowering sampling rate, the effect of burst attack in changing overall traffic pattern can be reduced. Therefore, the attacker will have to inject traffic with high rate for a long time to have impact on the traffic pattern. It would be foolhardy for us to say that `npf`'s classification accuracy will not be affected by this dynamic sampling. However, we believe that this is a better way to defend the attack than complete loss of performance.

4.6 Performance Evaluation

`npf` is not designed for exploiting the rule-set characteristics. Nevertheless, we evaluate the performance of `npf` using both real world and synthetic rule-sets. We used 5 real world Snort rule-sets [42] in which each entry contains a 5-tuple (source and destination IP prefixes, source and destination port numbers, and protocol). We used the real world rule-set as generators to produce 5-dimensional synthetic rule-set. The synthetic rule-set were generated using a simple technique. To create a new rule, we randomly pick the source and destination port and protocol from a pool of all the values that were in the corresponding 5 real world rule-set. We generate randomly source and destination IP prefix. This procedure is then repeated N times to create a rule-set of size N. Both type of rule-set size ranges from 25 to 10^3.

4.6.1 Metrics

We consider the number of operations performed per packet as the primary metric in order to analyze the performance of `npf`. We take the average on the number of operations performed per

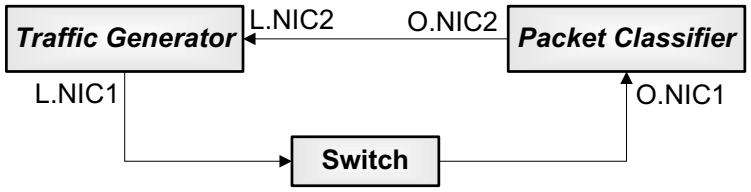


Figure 4.5: A simple view of the experimental test bed

packet to make sure that the system will perform as expected and the performance will not drop to an unacceptable level with the varying traffic distribution.

4.6.2 Experimental Test bed

The performance of our proposed approach in a real network configuration is demonstrated by writing a new packet filter using NetBSD's `pfil` framework [43] as an IDS on a NetBSD [44] machine. The testbed in Fig. 4.5 contains two machines: a traffic generator and traffic-adaptive packet classifier (3 GHz CPU and 2 GB memory). The generated traffic is forwarded to the traffic-adaptive classifier. If the packet matches a rule with "accept," the classifier forwards the traffic back to the generator, otherwise blocks it.

4.6.3 Experimental Results

Several Internet packet traces were captured at different machines of our lab and then replayed back using `tcpreplay` [45] in the experiments. The traces contain packet header information for 2 million to 5 million packets. The packet headers were captured at different days of week and times of day. It is reasonable to assume that these traces reflect realistic network conditions. The packet frequency distribution of two of the packet traces used in the experiment is shown in Fig. 4.6(a) and 4.6(b).

In Fig. 4.7, we plot the average path length (average number of operations) taken by `npf`

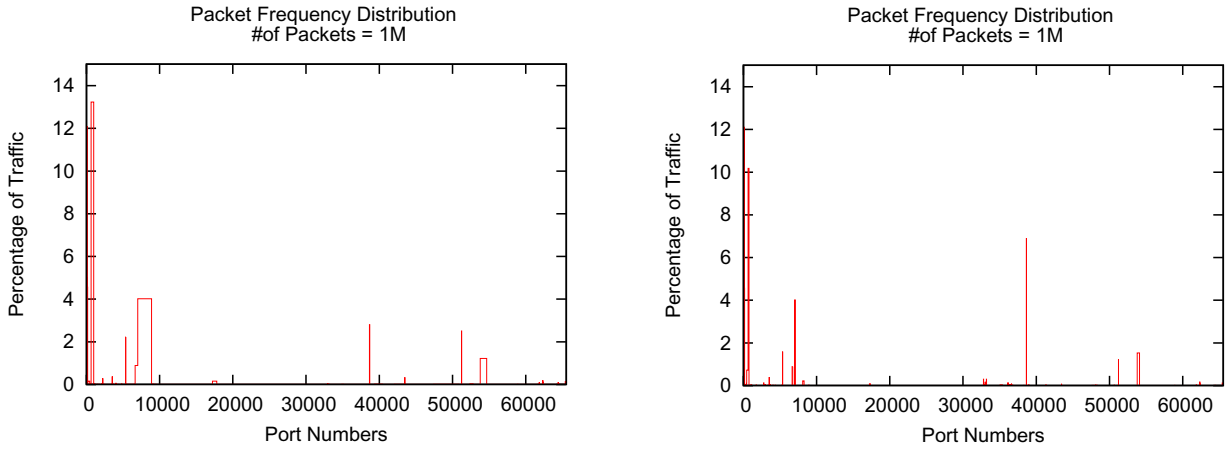


Figure 4.6: (a) Packet frequency distribution (Type-1) (b) Packet frequency distribution (Type-2)

per packet. The average is taken over the whole packet trace, with the tree updated online along the change in the traffic pattern. The graph shows attractive measurement that `npf` can classify packet with less than four evaluations per packet. We also calculate the entropy (theoretically) which is a measure of the smallest number of evaluations possible per packet for a particular traffic distribution. Fig. 4.7 shows a performance gain in `npf` (< 4 evaluations) compared to the calculated entropy of 5.345 evaluations per packet because of the inherent advantage of our underlying data structure that the node associated with the most frequently visited rules can be placed at the root of the tree which has 0 path length. Also, our tree building technique allows us to place the intervals not only in the leaf nodes but also in the internal nodes. As a result, our approach consumes less number of memory accesses (i.e. less number of operations) on average compared to the theoretical limit (calculated entropy).

We take measurement to find out the gain achieved in packet classification time due to exploiting traffic pattern. Fig. 4.8 shows that the performance gain was in an attractive range of 36–61% approximately.

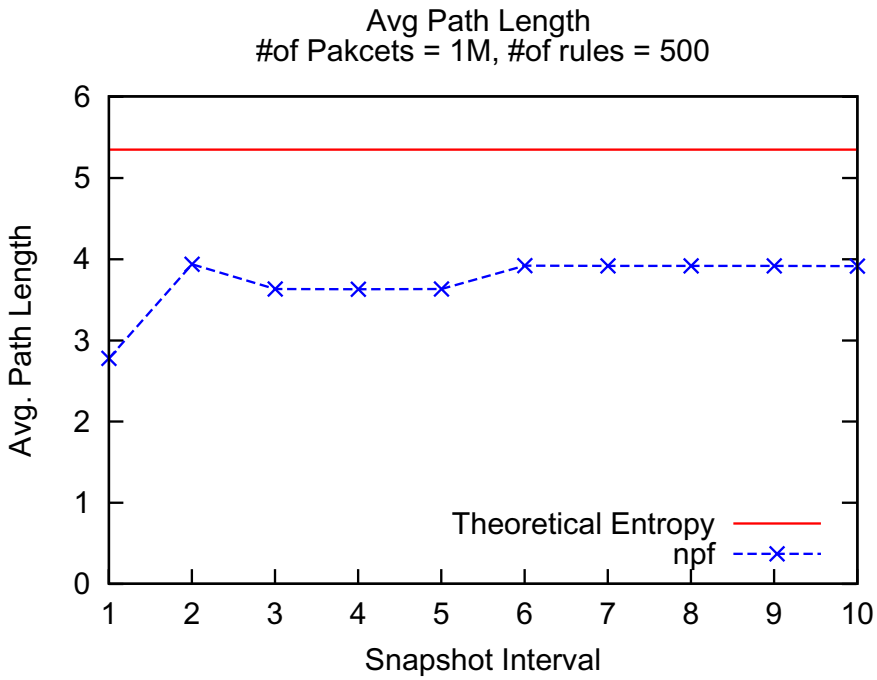


Figure 4.7: Actual cost per packet in number of evaluations using `npf`

The reason of consuming small number of operations per packet is because we promote the nodes associated with the most frequently visited rules closer to the root. However, it may happen that for a particular unpopular rule, our approach may need to take longer path if the node associated with that unpopular rule is located in the bottom of the tree. Hence, we take a measurement to see the trend of maximum tree height in our experiment. Note that we started with balanced binary tree which is the minimum possible tree height for a particular traffic distribution at the beginning of the experiment. Fig. 4.9 shows that the tree height increases from the minimum possible height as the experiment progresses and becomes stable as our approach adapts to the pattern of the traffic. Although the maximum tree height is longer (≈ 21) in this result, our approach can classify packet with less than 4 evaluations (Fig. 4.7) because the infrequently visited rules reside in the bottom of the tree and those are accessed less compared to the popular rules which reside closer to the root

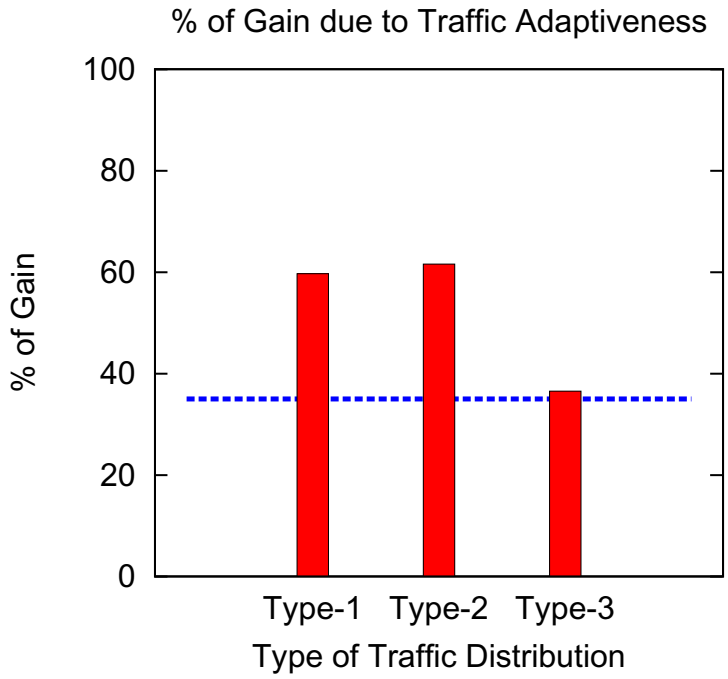


Figure 4.8: Gain in packet classification time by exploiting traffic pattern for three different traffic distribution (Fig. 4.6(a) and 4.6(b))

of the tree.

Finally, In order to justify the claim that running 'online adaptation' achieves better performance gain than 'offline,' we compare `npf`'s online adaptation approach with Adel et al. [1] approach in which the tree organization (adaptation) is done offline at certain interval. We conduct six different experiments with different traffic distribution to find out the effect of online vs. offline adaptation on classification time. In 3 different types of experiments, segment weights (section 4.3) are assigned using traffic of previous interval which are *E-1*: traffic distribution of current interval is kept same as previous interval, *E-2*: traffic distribution of current interval is made opposite as previous interval, *E-3*: traffic distribution of current interval is made randomly different than previous interval. In other 3 different types of experiments segment weights are assigned randomly which are *E-4*: traffic distribution of current interval is kept same as previous interval, *E-5*: traffic

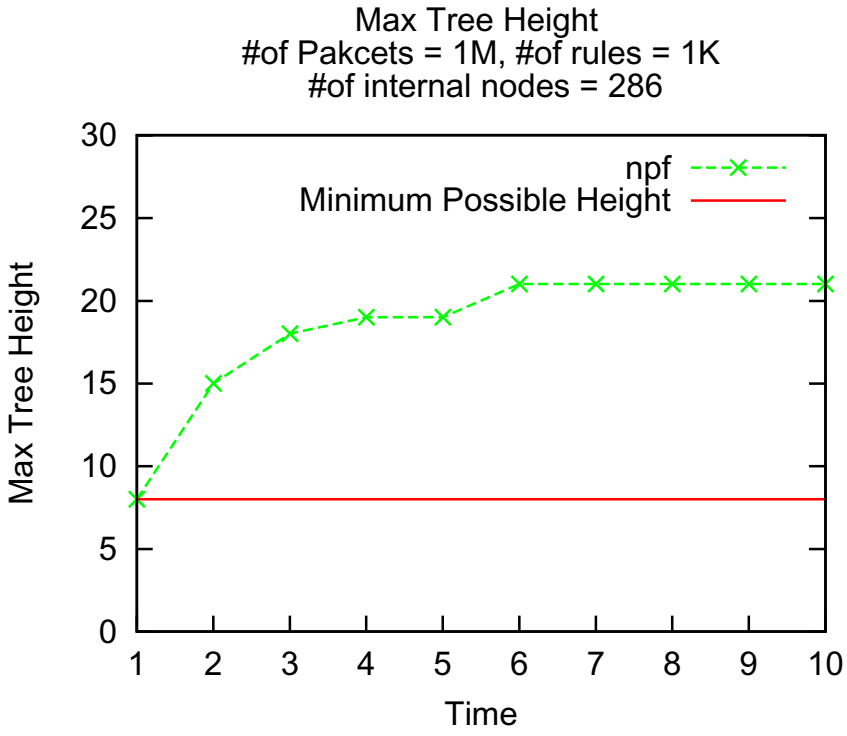


Figure 4.9: Maximum tree height of `npf`

distribution of current interval is made opposite as previous interval, and *E-6*: traffic distribution of current interval is made randomly different than previous interval. Fig. 4.10 shows that in all types of experiments, packet classification time was reduced more than 75% in online adaptation approach compared to the offline adaptation approach. This large reduction is attributed to two inherent features available in `npf` - first of all, due to online adaptation most frequently visited nodes reside closer to the root and second as soon as our algorithm finds that the result of $\cap$ operation between two dimension is $\emptyset$ (section 4.4.2), it bypasses the search operation and $\cap$ operation for other dimensions and classify the packet faster (incurring less memory accesses).

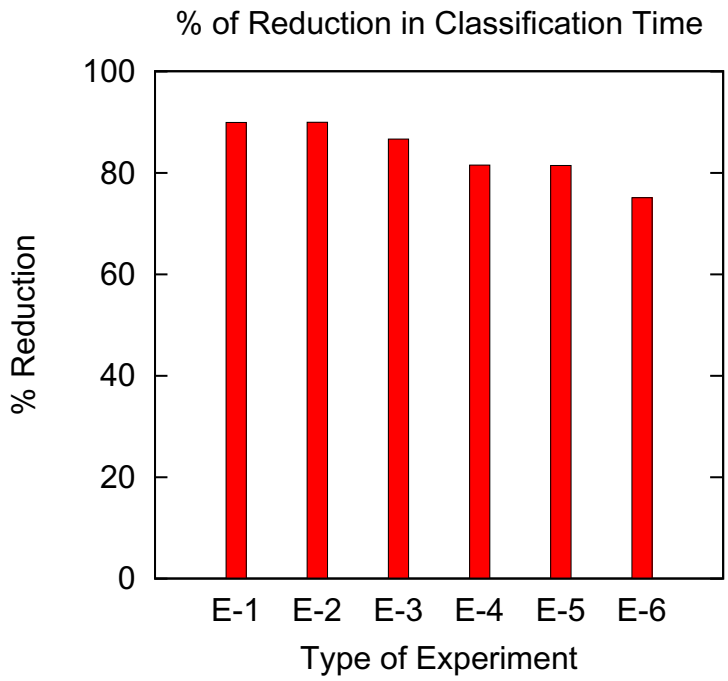


Figure 4.10: Effect of online adaptation (`npf`) vs. offline adaption [1] on packet classification time

Preprocessing Time Complexity

During preprocessing stage, `npf` converts the rules in the rule-set into intervals, which takes $O(ni)$ where n is the number of rules and i is the number of intervals. The `IRT` construction time is $O(n \log n)$.

Space Complexity

Fig. 4.11 demonstrated the memory cost of the tree search structure (`IRT`) as a function of the number of rules. The memory cost appear linear growth without requiring any significant additional space. `npf` belongs to the algorithm "trade space for time", it use a lower speed but large capacity and cheap RAM to achive comparative performance as the expensive and power-hungry hardware (such as FPGA, TCAM etc.)

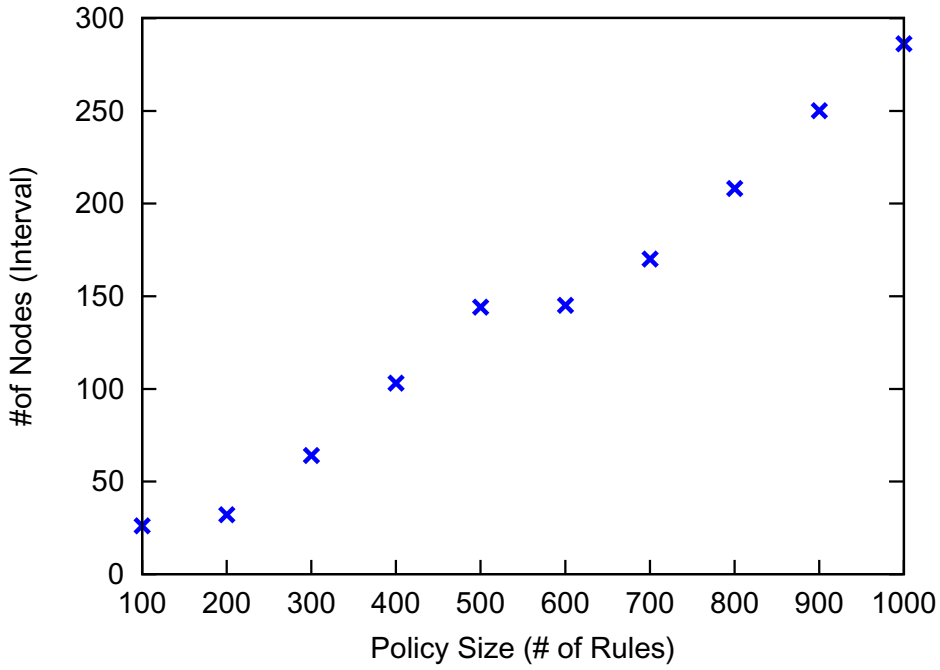


Figure 4.11: Number of interval (nodes) in IRT

Overheads due to Online adaptation

In order to exploit traffic pattern, we update tree dynamically (Section 4.4) to promote the nodes associated with the more frequently visited rules closer to the root. Online IRT update reduces the search time for a matching node for the most frequently visited rules. We have found in our experiments that this extra overhead update operation is extremely useful to achieve relative gain (Fig. 4.8) in search time on average if the traffic shows "skewness" (Section 4.1). In Fig. 4.12, we see that although the number of tree update is large at the beginning, it converges to a stable situation as our approach adapts to the pattern of the current traffic. The data for this graph was taken at several intervals during the experiment to show how our approach adapts to the pattern of the traffic over the time. However, we are curious to know what happens when IRT is updated too frequently. Does the processing overhead caused by unnecessary adaptation pose a performance

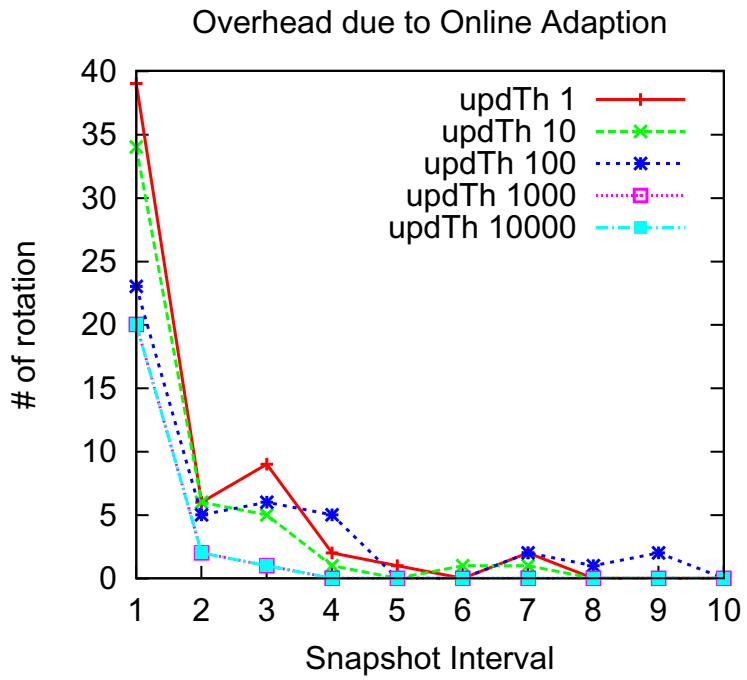


Figure 4.12: Number of rotations performed over the time

issue? Fig. 4.12 shows how the `IRT` update rate can be controlled by employing a update threshold. In particular, if `npf`'s performance degrades to an unacceptable level (performance level can be measured from the average number of evaluations per packet Fig. 4.7) enforcing a higher update threshold would reduce the tree update rate and thus avoid the unnecessary adaptation. However, enforcing a higher update threshold would affect the classification time. To see how the change in update threshold affect the classification time, we plot, in Fig. 4.13, the percentage of increase in classification time for different update threshold compared to the classification time when update threshold is set to 1 (best case). We see approximately 15% increase in classification time when update threshold is increased from 1 to 10^4. This is acceptable than trashing the system because of excessive tree update.

Since online tree update is crucial for achieving the gain, we are curious to find out how

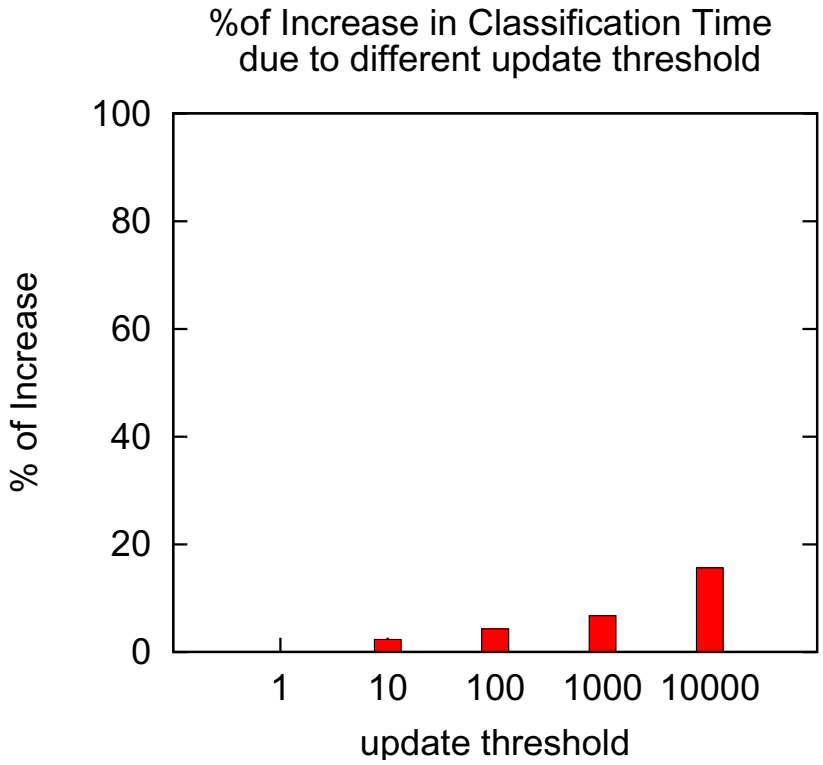


Figure 4.13: Effect of different update threshold in packet classification time

much processing overhead is introduced by the online dynamic adaptation of data structure? we took measurements to find out the time needed to modify the tree structure in real time. Our experiments with different traffic distribution (Fig. 4.6(a) and 4.6(b)) shows that approximately 4–6% of total classification time is spent for modifying the tree structure in real time.

4.7 Conclusion

Most of the current packet classification algorithms reported in the literature do not consider the traffic pattern in optimizing their techniques. In this work, we undertook the investigation of finding the feasibility of exploiting the locality in traffic to improve the average performance of packet classifier. We show that classification time can be reduced by performing online dynamic adap-

tation to the incoming traffic pattern. In particular, our test bed based analysis shows the tradeoff between the performance gain and processing overhead due to dynamic adaptation and a comparative study between online and offline adaptation approach. It is evident from our experimental results that the dynamic adaptation to the traffic pattern is very effective in boosting packet classifier's average performance because the most frequently visited rules can be found by incurring less memory accesses (i.e. spending less time per packet) since heavily hit nodes will remain in cache and as a result the cache hit ratio will be high.

CHAPTER five

PARALLEL PACKET CLASSIFICATION SYSTEM

In decision-tree-based Packet Classification (PC), packets are classified by searching in tree data structure. However, tree search presents significant challenges because it requires a number of unpredictable and irregular memory accesses. Packet classification is per-packet operation and memory latency is considerably high. The growing trend of number of rules in the classifier coupled with the constant increase in link speeds makes wire-speed classification a challenging task. Hence, satisfactory performance of PC still remains elusive at the wire speed. In this chapter, we present a parallel traffic-aware classification approach in high-end multi-core architecture available today. In particular, we present the design, implementation, and evaluation of traffic-aware classification system that exploits the locality in traffic patterns and leverages the task-level parallelism on multi-core general purpose processors to achieve Gbps classification speed.

5.1 Introduction

Packet classification is primarily used by network devices, such as routers and firewalls, to do additional processing for a specific subset of packets. Such additional processing includes packet filtering, quality of service (QoS), and differentiated services (DiffServ). Packet classifier categorizes packets based on a set of rules that represent the classification policy. Most of the existing packet classification solutions reported in the literature exploits the characteristics of rule-sets in optimizing their schemes. However, the prior observations [2, 10, 12, 20–22] that a relatively small subset of rules are visited more frequently in the rule-set show promise to another direction that packet classifier's average performance can be improved by exploiting the locality in the traffic pattern.

We designed npf [36], a lightweight traffic-aware packet classification system that achieves superior average performance by exploiting the traffic pattern. For each header field (source and destination IP, source and destination port, protocol), npf creates a special data structure called "Interval based Rule Tree (IRT)" in order to organize the rules in the rule-set. During classification, IRT is reorganized dynamically so that most frequently used field values are located at the shortest path in the search tree. The performance gain in npf is achieved when a large fraction of packets are classified by visiting the top part of corresponding header field's IRT incurring less number of memory accesses on average.

Packet classification is per-packet operation and for packet classification, the time to process a packet is dominated by memory-access latencies. Hence, an architecture containing a single, high-performance processor is often not suitable for a classification system. In addition, the growing trend of large rule-sets size coupled with the recent advancement in transmission link rates (Gbps) makes wire-speed classification a challenging task. To mask memory access latencies and to meet the performance demands, and thereby process packets at high rates, it is important to exploit the inherent parallelism of network processing. Most packets in network traffic do not exhibit interdependencies and thus can be processed independently. As a result, modern multi-core general purpose processor can be utilized to employ highly parallel architectures for packet classification. The increased computing power, inexpensive system cost, easy programmability of multi-core processors have lead researchers to employ them for high-speed packet processing [46–48].

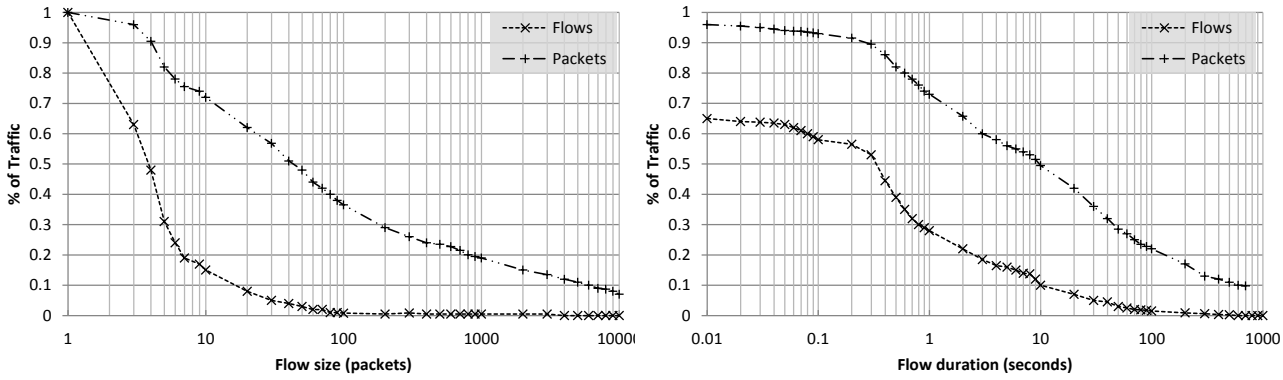
In order to achieve the required high throughput, we pursue to make strong use of parallelization. In this work, we explore how the highly parallel capabilities of commodity multi-core processors can be utilized to further improve the performance of traffic-aware packet classification system. An important property of npf is its modular nature. As a result, npf is a "natural" fit for multi-core technologies, because npf can be visualized as a set of tasks (or modules) and the

tasks can be assigned to distinct cores to run independently in parallel. In this work, our effort is to take advantage of this modular nature of `npf` with the use of multithreading and with multiple execution cores on a general purpose multi-core architecture.

We have designed, implemented and evaluated `Pnpf`, a parallel traffic aware packet classification system on modern multi-core architecture. The design of `Pnpf` (**P**arallel `npf`) is a multi-threaded software-only solution that accompanies a load balancing mechanism which balances the throughput of its multiple constituent components, such that the overall throughput of the entire system is maximized. We use pipeline model (i.e. task-level parallelism) in `Pnpf`. Here, classification is partitioned into multiple stages, stages are connected together in sequence, and each processor is assigned a specific stage to execute. Small queues (with exclusion locks) reside between adjacent stages to decouple read and write operations and provide for a limited amount of buffering. Compared to data parallelism, this approach offers reduced lock contention (since each lock is shared by at most two processors), reduced latency, and good cache locality. While our design of `Pnpf` is implemented and evaluated for a popular multi-core architecture, this software-only solution can be embedded in any other platform, for example Cavium's family of Octeon-based network processor boards [49].

5.2 Internet Traffic Properties

The design of `npf` is highly motivated by the Internet traffic properties that were observed in our study [36] and addressed by other researchers as well [2, 10, 12, 20–22]. After studying publicly available Internet traffic and private traffic traces we find out that the majority of network traffic/flows matches a small subset of the rules in the rule-set. Another key finding in our observation is that this "skewness" property in the distribution of network traffic continues for long enough time intervals such that it is reasonable to exploit such skewness to improve the average performance



(a) (b)

Figure 5.1: Distribution of a) flow size, and b) flow duration [2] for packet traces available at [3].

of packet classification. Many of the current packet classification techniques does not consider the traffic pattern in their optimization techniques; rather they exploit the characteristics of rule-set. On the other hand, `npf` considers the traffic pattern to improve its average case performance.

In this section, we will present detail analysis on publicly available traffic traces [3] to demonstrate that the majority of the incoming (or outgoing) packet is matched against a small subset of rules in the rule-set (skewness of the traffic matching the rules) and traffic skewness property is unlikely to change over a short period of time. We perform the traffic analysis on the packet traces available at [3]. The traces contain packet header information for 2 million to 10 million packets. The packet headers were captured at different days of week and times of day. It is reasonable to assume that these traces reflect realistic network conditions.

5.2.1 Packet flow properties

As shown in Figure 5.1(a) around 20% of the flows have 10 packets or more and around 60% of the flows have 3 packets or less. Also, around 70% of the Internet traffic are carried by the long flows. Next, we look at the distribution of flow duration. As shown in Figure 5.1(b), about 20% of the flows (that carry about 60% of the total traffic) have a duration of 5 seconds or more

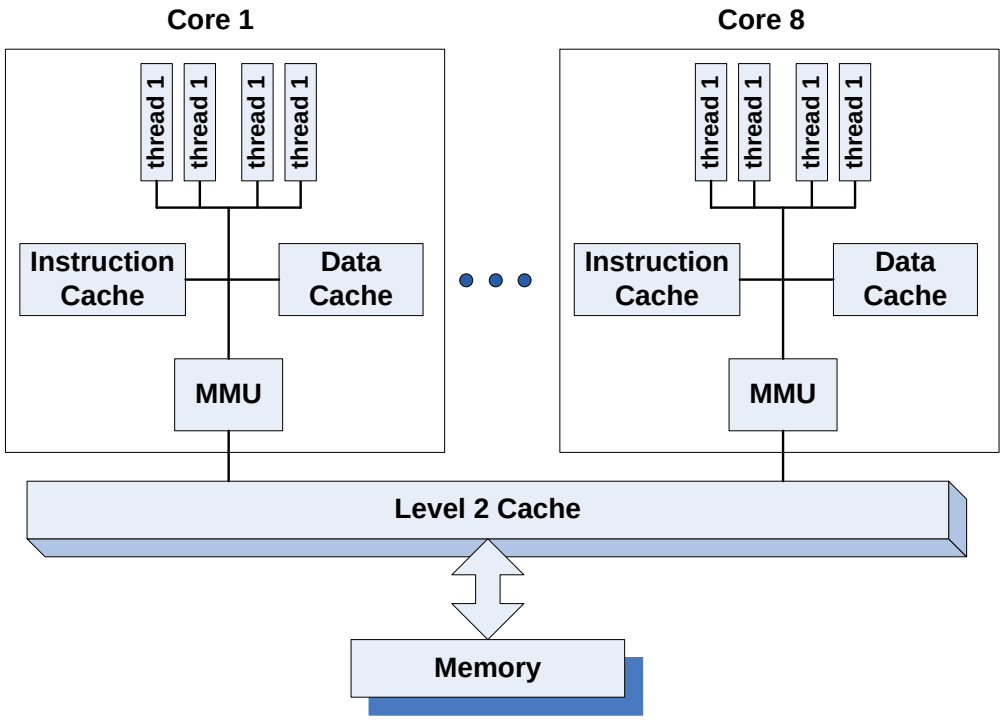


Figure 5.2: An overview of multi-core architecture

(i.e. long-lived flows carry most of the traffic). These observations imply that a significant portion of the traffic is matched against a relatively small subset of the rule-sets over extended duration. Our observation is consistent with the previous study [50] which reported that while the most of the Internet flows have short flow sizes, the considerable amount of Internet traffic is constituted from the long flows. As a result, this clearly indicates that exploiting traffic pattern to optimize classification performance is not only useful but also practical in most cases.

5.3 Introduction To Multi-Core Architecture

Modern designs include multi-core systems, where a single die holds multiple CPUs. Multi-core processors use chip multiprocessing (CMP). Rather than just reuse select processor resources in a

single-core processor, processor manufacturers take advantage of improvements in manufacturing technology to implement two or more "execution cores" within a single processor. These cores are essentially *N* individual processors on a single die. (Figure 5.2). Execution cores have their own set of execution and architectural resources. Depending on design, these processors may or may not share a large on-chip cache.

Commodity multi-core processors gained popularity [51,52] because they offer the raw parallelism necessary to address the problem, namely the incessant growth of network traffic volumes and rates. The low cost multi-core processors made it feasible to build complete packet classification system using general purpose processors. Architects and developers in the industry [53] are now considering these processors as an attractive choice for implementing a wide range of networking applications, as performance levels that could previously be obtained only with network processors (NPUs) or ASICs can now also be achieved with multi-core architecture processors, but without incurring the disadvantages of the former such as the requirement of highly deliberate, and customized programming. Today one can buy quad-core [54], six-core [55], and 8- core with 8 threads/core [56] CPUs. These designs promise to continue scaling into the future; for example, there are already specialized 64-core processors for network processing [57] and 86-based many-core architectures that may contain 64 discrete 86 cores with vector extensions [58].

While there are many ways to take advantage of the computing power of multi-core systems (ex. virtualization, SMP OS, etc.), multi-threading individual applications may be the best way to realize large performance gains on such platforms. Chip multiprocessors (CMPs), exploit thread-level parallelism (TLP) and programmers must exploit thread-level parallelism in order to extract performance from CMP. CMP is arguably the most common architectural method for networking applications. The reason for this selection is motivated by the nature of the networking applications. Most networking applications exhibit task and data level parallelism. In addition,

most applications are relatively simple. Therefore, designers utilize several simple execution cores that can take advantage of the data/task level parallelism without complicating the design process. CMP can improve packet processing performance by executing concurrent processes/threads on different execution cores. Consequently, to provide more computing power and to match I/O processing required by high bandwidth network, CMP are widely used as server platforms. For example, multi-core architectures are employed in Cisco's Application Oriented Network (AON) technology [59].

We choose to deploy our parallel packet classification system in CMP multi-core environment because of the following reasons:

1. ease of development (As opposed to NPUs, the software engineers are not required to learn a special purpose programming model and tools for packet processing)

2. promise of long term support from the major chip manufacturers [53].

3. CMPs natural tolerance for inter-thread communication optimizes the necessarily small computation time per packet, and does not mandate the processing of many packets at a time to provide scalability [30].

5.4 Architectural Paradigms

This section explores different programming models that can be used for developing `Pnpf` and their influence on the performance of `Pnpf` in the multi-core architecture processor environment. In order to deal with high data-rates, several architectural paradigms have been commonly used:

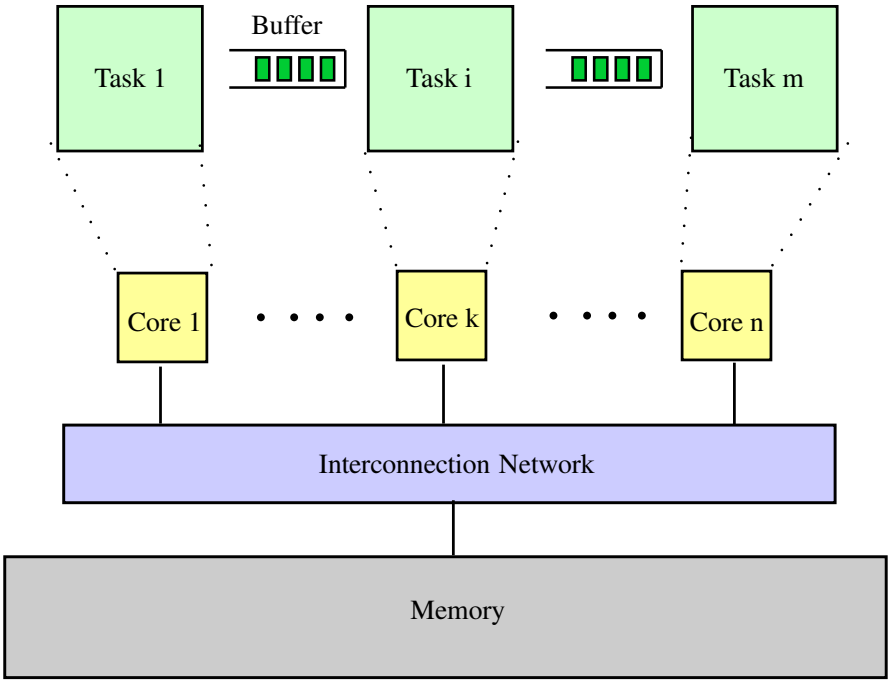


Figure 5.3: Pipeline architecture in which each incoming packet flows through multiple stages of a pipeline.

5.4.1 Parallel processing with multiple processors

In this model, a packet is processed in a single step. This the simplest approach to parallelize a workload. The processing loop is entirely replicated for every thread. Threads are well-supported models of concurrent programming. However, they often entail high overhead in terms of context-switch time and memory footprint, which limits concurrency.

5.4.2 Pipeline of processors

This model (Figure 5.3) comprises a number of steps, called pipeline stages. Each stage of the packet classification pipeline is mapped to a different core/thread, with the packet being sent from one stage to the next one in the pipeline. Each core/thread has its fixed place in the pipeline and is

the owner of a specific functionality which it applies on a single packet at a time, so the number of packets currently under processing is equal to the number of pipeline stages. The pipeline model has the advantage of offering an easy way to map the packet processing pipeline to the computing resources of the processor by simply assigning each pipeline stage to a different core/thread. Better throughput can be achieved if the code footprint and the data working set of a given pipeline stage is similar for most of the packets and dependencies with other cores can be kept at minimum.

5.5 Challenges in Multi-threaded design on CMP architecture

No increase in the processing power is achieved by simply using the multi-threading. Multi-threading cannot minimize the latency of complex operations, but it can be an effective mechanism to hide this latency from the cores and thus increase the overall efficiency of the cores. However, the multi-thread model is not problem-free, as it introduces the delicate problem of synchronization between the cores/threads when accessing the shared resources such as the input/output packet streams, the shared data structures, etc.

A general rule of thumb in multi-threaded application is to share as little data as possible between threads because the synchronization overhead can really limit performance. In general, it is recommended to design the data structures of the application for minimal contention between the cores/threads accessing them.

Also, it is very important to distinguish read-only resources from shared modifiable resources to address the problem of data race. Data race is not a concern or does not occur when multiple threads simply attempt to read a block of resources simultaneously or access simultaneously a resource that cannot be modified (that is, read-only memory or const objects). In order for a race condition to exist, the resource under consideration must be modifiable, and multiple threads must be trying to simultaneously access the resource with at least one of the threads attempting to

modify the resource.

Communication between threads or synchronization between processors comes at a cost. The complexity of the synchronization or the amount of communication between processors can require so much computation that the performance of the tasks that are doing the work can be negatively impacted. We consider the above mentioned issues while designing the parallel pipeline version of `npf`.

5.6 Design of `Pnpf`

In this section we briefly discuss about `npf`, a light weight traffic-aware packet classification system. Then we present the design of `Pnpf` (parallel `npf`), which is designed to enable high concurrency with load conditioning on CMP architecture.

5.6.1 Introduction to `npf`

Table 5.1: An example of rule-set

ruleID	SrcIP	DestIP	SrcPort	DestPort	Protocol	Action
1	*	*	[600,610]	80	TCP	Accept
2	*	*	[1000,1024]	[1024,65535]	TCP	Reject

Most of the previous researches on packet classification focus on optimization based on deterministic techniques. On the other hand, we focus on considering the statistical properties of the traffic passing through the classifier as an optimization technique to achieve a near-optimal searching time. We propose `npf` that is different from the existing packet classifiers in two ways: first, it classifies traffic exploiting the locality in the traffic pattern and second, unlike existing traffic-aware approaches requiring a separate, off-line reorganization phase, `npf` performs reorganization

on-line with little overhead. npf is a decision tree based algorithm. It uses a special data structure called "Interval based Rule Tree (IRT)" in order to efficiently organize the rules in the rule-set. During classification, IRT is reorganized dynamically so that most frequently used field values are located at the shortest path in the search tree. This results in quick matching significantly for the most popular traffic. The dynamic adaptation to the traffic pattern in npf is achieved by using a technique called "tree rotation". Applying this technique, the nodes associated with the frequently visited rules are moved closer to the root to reduce the number of memory accesses for the popular rules. This simple technique results in extremely good performance in packet classification because it is highly likely that the next couple of searches for a matching rule will be found at the top part of the tree, thereby incurring less number of memory accesses because the nodes associated with the frequently visited rules is expected to be found near the root of the "IRT". In best possible case, our npf will find the matching rule for the most popular traffic in at most one step.

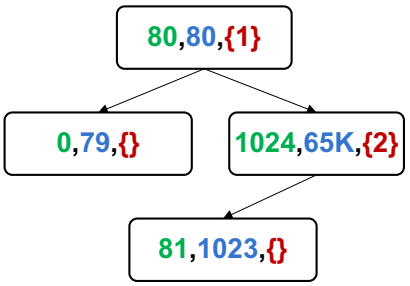


Figure 5.4: A typical "IRT" for Destination Port field in ruleset (Table 5.1). The legend of the node is as follows: begin of interval, end of interval, {list of rule IDs that contains this interval}.

We build IRT for each dimension (field) in the rule-set. The tree structure is named as "Interval based Rule Tree" because everything (IP address and port number) is considered in terms of interval or range. For example, 16 bit port field ranges from 0 to $2^{16} - 1$. Thus the port field has an interval of $[0, 65535]$. Figure 5.4 shows a typical IRT for the destination port field in Table 5.1.

We also convert the IP prefix into appropriate interval. The detail of `npf` can be found in Chapter 4.

5.6.2 `Pnpf` Design

Due to the inherent parallelism in network traffic, the design space for parallel packet classification system is vast and ranges from pipelined to multiprocessor solutions. Most packets in network traffic do not exhibit interdependencies and thus can be processed independently. Therefore, it is very important to exploit the inherent parallelism of network processing in order to meet the performance demands of a packet classification system at high link speed.

Parallelizing any application requires first identifying the available concurrency. Concurrency is exposed by using data parallelism or task parallelism or combination of both. In data parallelism model, identical tasks process different set of packets independently and in parallel. In task parallelism, the entire work load is partitioned into components that are completely independent of each other. These components are then scheduled to execute in parallel. It is critical to leverage the right combination of data and task parallelism because the gains from parallel execution can be overshadowed by the costs of communication and synchronization.

We exploited one type of task parallelism, pipeline model to parallelize `npf`. This paradigm has a number of benefits: first, it does not require any synchronization or lock mechanisms since different cores/threads process different data in isolation. Second, having several smaller data structures instead of sharing a few large ones reduces the size of the working set in each cache, increasing overall cache efficiency. In addition, pipeline model provides deterministic communication, following a producer-consumer pattern between pipeline stages and parallelism can also be exploited within stages using either data parallelism or multiple worker threads. However, one limitation of pipeline model is that applications parallelized using this model are very sensitive to the load balancing across the stages. Our design includes controller that dynamically balances load

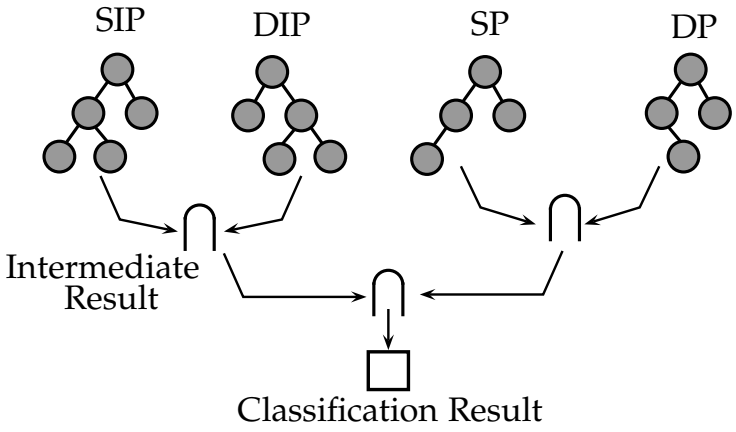


Figure 5.5: `IRT` tree structures for parallel matching for $d = 4$ dimension packet classification. "SIP" means Source IP, "DIP" means Destination IP, "SP" means Source Port, "DP" means Destination Port.

across the stages to ensure an overall good throughput (Section 5.6.4).

Task level parallelism seems a natural granularity for parallelizing `npf`. In `Pnpf` (the parallel `npf`), we partition the entire packet classification system into components that are completely independent of each other. Multiple `IRT`'s are constructed because packet matching is performed on multiple fields (5-tuple source/destination IP, source/destination port and protocol). These independent tree structures (Figure 5.5) are suitable in pipeline model on a multi-core environment because tree traversals over header fields offer independent operations which can be performed concurrently on distinct processors. All these trees, five in our case, are searched in parallel and the matching results of the independent field searches are then combined to produce the final classification results.

As shown in Figure 5.6, `Pnpf` is decomposed into seven pipeline stages. The first and the last ones are stages for input and output. These are the serial stages that read a set of network packets to be classified against a rule-set, and output the highest priority ruleID that matches the packet, respectively. The middle 5 stages are parallel and configured with a thread pool of size t

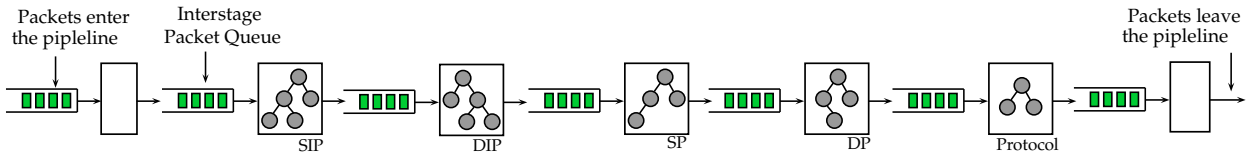


Figure 5.6: `Pnpf` pipeline configuration. This is a structural representation of `Pnpf`. The parallel classifier is composed as a set of stages separated by queues. Edges represent the flow of packets between stages. Each stage can be independently managed, and stages can be run in parallel. The use of packet queues allows each stage to be individually load-conditioned. "SIP" means Source IP, "DIP" means Destination IP, "SP" means Source Port, "DP" means Destination Port and "proto" means protocol.

where $t \geq 1$. Packet data is passed between stages using software queues, configured to hold 1000 entries (default *Queue_size*).

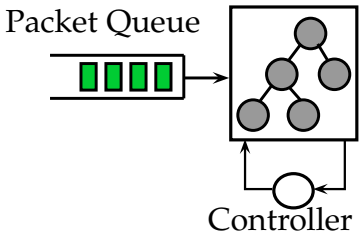


Figure 5.7: A `Pnpf` Stage: A stage consists of an incoming packet queue, a thread pool, and tree structure `IRT`. The stage's operation is managed by a controller, which dynamically adjust resource allocations.

The fundamental unit of processing within `Pnpf` is the stage. A stage is a self-contained application component consisting of a tree structure (i.e. `IRT`), an incoming packet queue, and a thread pool, as depicted in Figure 5.7. Each stage is managed by a controller that affects thread allocation. Threads within a stage operate by pulling a batch of packets off of the incoming packet queue and traversing the tree structure. After the tree traversal is complete, the packets and the set of ruleID that matches the current header field are enqueued on the packet queue of the following stage. Each packet flows through the entire pipeline, and a given stage of the pipeline performs part of the required processing. An arriving packet enters the first stage, which looks up the source

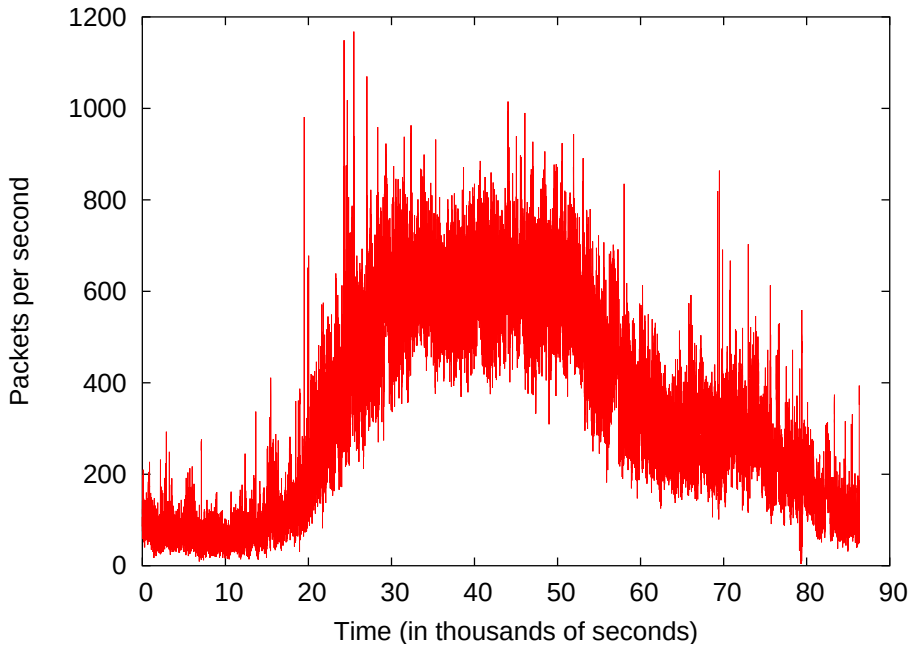


Figure 5.8: Arrival of Packets (per second) over a day for the NLANR trace [3].

IP part of the packet header. The later stages perform lookup on other header fields of the packet header. When the packet reaches the last stage, the lookup results (sets of ruleID) are combined to find the matching rule for this packet. Because each stage is executed by independent core/thread, all stages can operate at the same time. Of course, parallel execution of all stages only occurs when packets arrive in rapid succession (Figure 5.8); a pipeline remains full only if a new packet is ready to enter each time a packet leaves. We expect that the average time spent for tree search on different stages will not significantly deviate because the tree structure on different stages dynamically adapts to the traffic pattern. However, in practice the time required to lookup can vary across the stages. We therefore, employ packets queues between stages. Thus, a given stage can place a packet in an output queue and begin working on the next packet, even if the succeeding stage remain busy. The use of queues has a number of benefits, including isolation, independent load management, and code modularity. Introducing a queue between two stages decouples their

execution, providing an explicit control boundary. The execution of a thread is constrained to a given stage, bounding its execution time and resource usage to that consumed within its own stage. A thread may only pass data across the boundary by enqueueing a packet for another stage. As a result, the load balancing of each stage can be controlled independently.

Contention for locked resources usually becomes one of the major bottlenecks to multi-threaded application performance. Another motivation for choosing pipeline model is that assigning task specific resources to single stages ensures less access conflicts and reduced communication effort. Access to data structures shared across multiple packets only needs to be synchronized on the threads running on single core. In `Pnpf` the tree structure is reorganized dynamically to adapt to the pattern of the network traffic. By isolating each tree structure into distinct stages reduces the amount of communication and resource sharing among the cores/threads.

In CMP architecture, each core has its own memory cache. Thus, we can achieve good memory performance by scheduling threads that share the same data structure onto the same core, while executing unrelated threads on another core. As the performance gap between cache and main memory increases, improving cache locality is critical to yield performance gains. Another benefit of core dedication is that the possibility of cache collision is very small. Therefore, it is much better to always schedule the task on the same core to minimize overhead like cache misses or TLB flushes. Previous work on multi-core OSes also suggests that isolating applications into different cores outperforms symmetric scheduling because dedicating cores ensure an effective use of cache resources and a reduction in lock contention [60].

We also considered other design choices. One alternative design is shown in Figure 5.9. This design follows the data parallelism idea used by previous software router implementations [61, 62]. In this alternate design, we could take all the different task components of `Pnpf` and fold them into a serial sequence of tasks. Each core implements a full version of `Pnpf` that acts on

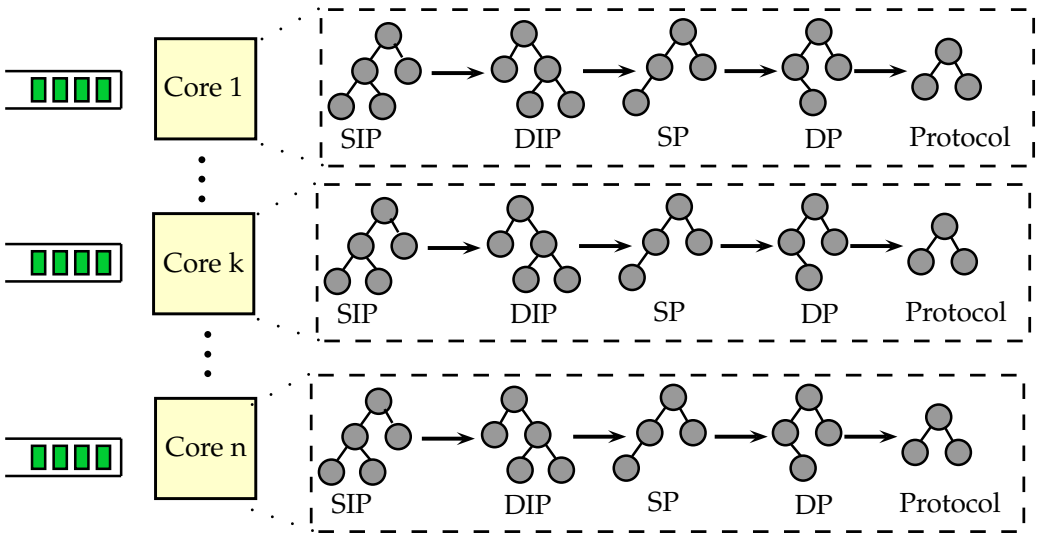


Figure 5.9: An alternative architecture that exploits data parallelism.

separate sets of incoming packets for data parallelism. However, we find that this model of parallelization of `npf` will not yield noticeable performance improvement due to excessive sharing and high lock contention cost associated with protecting shared data structures.

5.6.3 Packet Classification in `Pnpf`

Packets move through a series of stages along the pipeline (Figure 5.6) from one processing element (core) to another. Each instance of the pipeline passes packets from one stage to the next, across core boundaries. So at any one time, the entire pipeline may have multiple packets in various stages of processing. Packets are placed onto a ring buffer, which serves as a holding point for the remainder of the packet processing. Threads in the first stage removes packets from the ring and begins the processing the packet header. For IPv4, packet classification, this processing takes place on the Internet Protocol (IP) portion of the packet. In order to perform the search on multiple fields (5-tuples), packet lookup is performed against each of the field's search trees separately on different stages. We obtain for each field a set of candidate rules that contain the corresponding

field value. The last stage in the packet processing pipeline finds out the rule that matches the packet by getting the intersection between these sets of rules. If the intersection contains more than one rule, the rule with highest order (priority) is selected. If no rules are common, then the default action is returned.

5.6.4 Optimizations

The scalability of `Pnpf` is gated by the load imbalance between stages and by the input stage (first stage). In this section, we discuss solutions to address these limitations encountered in applications parallelized using the pipeline model.

Input stage optimization

Study [46] indicated that the CPU spends almost 100% of its time responding to packet receive interrupts i.e. the system bottleneck is in the packet capture module. In `Pnpf`, there is no dependence between different network packets, therefore, we can parallelize the input stage (the first stage in the pipeline). We divide the first stage into two stages, a serial one that receives the network packets and enqueues them in an intermediate queue and a second parallel stage that extracts packets from the queue for processing by the later stages.

Dynamic Load Balance

Load imbalance is a result of the different amounts of work in each stage in `Pnpf`. It is common to use a unique number of threads for all stages in pipeline. Although, we expect that the amount of work across the stages would be similar because the tree structures are adapted dynamically to the traffic pattern, it may happen that the amount of work between stages may vary over the time. As a result, if we continue to use unique number of threads for all stages, the stage with the larger

amounts of work will become a bottleneck. Large differences in the amount of processing across stages will result into an inefficient parallel classification system.

In order to enforce balance on the amount of work between stages, we employ load balancing mechanism to minimize the proportions between the largest and the smallest processing task. Instead of using a unique number of threads for all stages in pipeline, the number of threads attached to a pipeline stage are adjusted dynamically to make the pipeline flow smoothly, which not only solves the possible imbalance problem, but also makes the thread-level pipeline parallel model scalable.

Our load balancing mechanism employs resource control technique by tuning the number of threads executing within each stage. The objective is to avoid allocating too many threads, but still have enough threads to meet the concurrency demands of the stage. The controller periodically samples the input queue (default: once per second) and adds a thread when the queue length exceeds some threshold (default: 100 packets). If threads are idle for more than a specified period of time (default: 5 seconds), the controller will remove them from a stage.

We also considered other load balancing choices such as "Work stealing". Load balancing is achieved by keeping one active thread per core and assigning several work units to each thread. When a thread completes its assigned work, it queries the other thread for additional work (i.e. it will steal work from another thread [63]). Although, in general, work stealing is an effective technique to balance the amount of work between stages, it turns out a bad choice for our application. The main reason is that the tree structures in each stage are dynamically adapted to the traffic pattern. If we let the threads of another stage to steal the work, it introduces data structure sharing. Data structure sharing between two stages raises a number of concerns. Consistency of shared data must be maintained using locks or a similar mechanism; locks can lead to race conditions and long acquisition wait-times when contended for, which in turn limits concurrency.

5.6.5 Mutual Exclusion by Atomic Operation

Certain features are critical to creating efficient software pipelines. Atomic operations are useful in coordinating access to shared resources while low-latency inter-process communication mechanisms (software queue) move packet information between pipeline stages and help to control synchronization among threads. If multiple threads extract/place packets from/to the same queue, the queuing process must ensure that packets are inserted/extracted atomically. In order to guarantee this, mutual exclusion is needed. Mutual exclusion can be implemented by locking. We use the built-in function for atomic memory access available in *gcc* [31] to implement a lock in our design. A details description on using atomic variable to implement locking is presented in Section 3.4.1.

5.7 Related Work

Packet classification is per-packet operation and for packet classification, the time to process a packet is dominated by memory-access latencies. To mask memory access latencies and to meet the performance demands (Gbps link speed), and thereby classify packets at high rates, parallelism is the must-adopt solution. Recent work has looked at approaches to parallelizing packet classification itself.

The researchers mainly leveraged either data-level or task-level or combination of both parallelism in designing parallel packet classifier. Inspired by instruction-level pipeline structure, various thread-level pipeline approaches were proposed [64–66]. The idea of parallelizing packet classification application using pipeline model was mentioned by [67]. The authors claimed that by applying pipelining, their prototype was able to classify packets at the rate that exceeds the required rate for OC-192 (i.e. 10 Gb/s). However, the authors did not present any detail description on the parallelization scheme.

Randy et al. [30] parallelized two classic packet classification algorithms using both data- and task-level parallelism on multi-core architecture. Their results showed that the hardware constraints are mitigated by the parallelization scheme and vice versa, yielding near-linear speedups as the degree of parallelization increases. However, the authors mentioned that the performance improvement depends strongly on many factors, including algorithm choice, hardware platform and parallelization scheme.

Yaxuan et al. [68] utilized parallelization on multi-core Octeon machine to achieve packet classification in Gbps rate. However, the authors did not clearly mention which type of parallelism was exploited to utilize the multiple execution cores.

Cheng et al. [69] proposed scalable packet classification algorithm that can be efficiently implemented on a multi-core architecture with or without a cache. The authors studied the interaction between the parallel algorithm design and architecture mapping to facilitate efficient algorithm implementation on multi-core architectures. The authors effectively exploited thread-level parallelism on multi-core architecture to enable an efficient algorithm mapping.

Guo et al. [70] explored potential connection level parallelism in pattern matching, and proposed an affinity-based scheduler to enhance the scalability of multithreading. The authors assigned packets belonging to the same connection to the runqueue of the same thread, which is dispatched to a dedicated core on a multi-core server.

In [71], the authors parallelized by partitioning the rule-sets and running multiple decision tree-based classifiers in parallel. Each tree covers a distinct subset of the rules.

We exploit task-level parallelism to design, implement, and evaluate a packet classification system on multi-core architecture that also exploits the potentiality of traffic pattern to achieve a superior average case performance. The idea of exploiting traffic pattern to improve the performance is proposed in [2, 10, 12, 20–22]. Hamed et al. [2] presented cascaded tree structure for

Table 5.2: Configuration details of test bed

Name	*Value*
Processor	AMD Opteron 2379 HE
Core Count	8 (4 per processor)
Core Speed (MHz)	2400 (per Core)
L1 Cache Size (KB)	128 (per Core)
L2 Cache Size (KB)	512 (per Core)
L3 Cache Size (KB)	6144 (shared by 4 cores)

single-threaded processing and parallel tree structure for multi-core architectures. However, the authors' design and implementation of parallel algorithm lacks details in analysis.

5.8 Evaluation

This section presents and evaluates `Pnpf` that yields improvements in performance by exposing more parallelism.

5.8.1 Test bed Setup

The hardware platform for our test bed is Dell Poweredge 2970. This machine has two CPU sockets, each embeds a quad-core AMD Opteron 2379 HE, and 16 GB RAM running Linux OS. The detail configuration of the machine is given in Table 5.2. We used the C language and POSIX pthread-library [33] because of their simplicity and portability in our implementation.

5.8.2 Performance Evaluation

In this section, we evaluate the performance of the `Pnpf`. We used real-life packet traces [3] in our experiments. We generated different size rule-sets based on the traffic flow information in [3], such that the rules will exercise the effectiveness of traffic awareness of `Pnpf`. To create the rule-

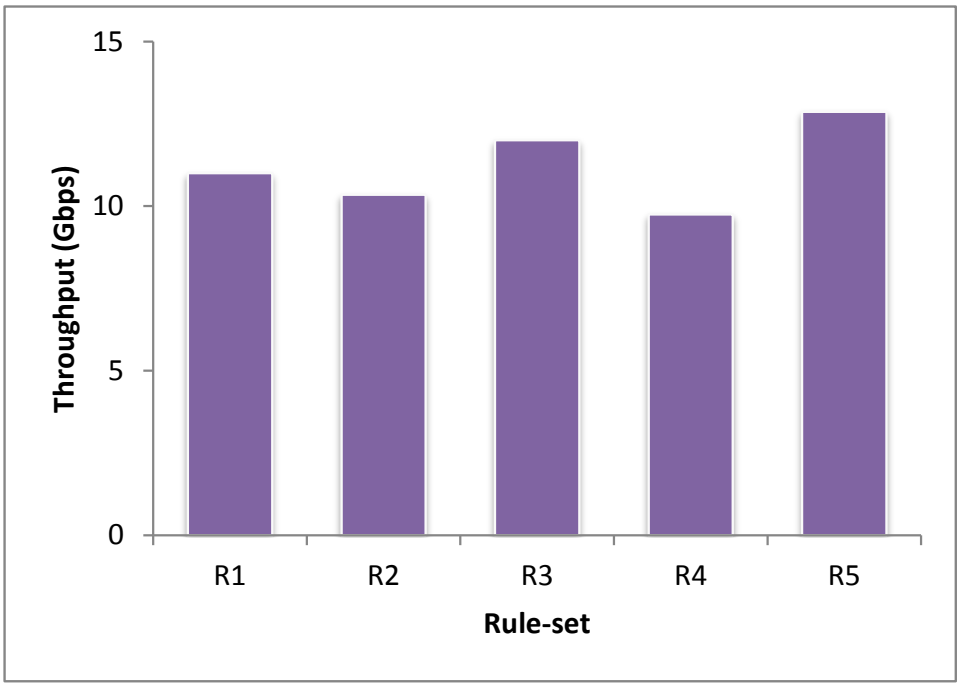


Figure 5.10: Throughput of `Pnpf` on rule-sets R1 through R5

set from [3], for each header field, we extracted 2^8 different values from the packet trace. The rules were generated using a simple technique. To create a new rule, we randomly pick different field values from the pool of all values in the respective header filed and then use them to form the 5-dimensional rule. This procedure is then repeated N times to create a rule-set of size N. All the rules in the rule-sets are 5 dimensional tuples composed of source/destination IP addresses, source/destination port numbers and protocol type. The size of the rule-sets ranges from 500 to 5K. We experimented with all rule-sets and present results from all of them. The packet size is assumed to be 128 bytes (1024 bits).

Figure 5.10 shows the throughput achieved by the `Pnpf` system where the x-axis represents the five rule-sets R1 through R5, and y-axis is the packet classification throughput achieved by the corresponding rule-set. `Pnpf` is able to achieve 10 Gbps or more in packet classification through-

put in almost all the rule-sets. The characteristics and the size of rule-sets do not have noticeable impact because Pnpf reduces the classification time by adapting to the traffic pattern dynamically. This online adaptiveness enable Pnpf to achieve higher throughput on average particularly when the traffic pattern is stable for an extended period of time (Section 5.2).

We must also note that pipeline parallelism is beneficial for data-intensive applications such as Pnpf, because compared to data-level parallelism, it reduces the contention on shared resources. In fact, we have compared our Pnpf with a programming scheme where all the cores are executing the full application. Our results indicate that for a 6-core system running the Pnpf application, the throughput of pipeline model Pnpf is 39.8% higher than the throughput achieved with replicating the Pnpf on each core. This is because synchronization and locking costs outweighed the benefits of concurrency.

We next study the scalability of Pnpf by varying the number of cores available in the system. We used the pthread_setaffinity_np function to bind all threads to a fixed number of cores. We varied the number of cores between 2 and 6. Figure 5.11 shows the speedup on the five different rule-sets. The baseline configuration for this experiment is a configuration where the entire application is run in a single core. Pnpf demonstrates a near linear scalability growth. We expect that this linear growth in performance can continue for a much larger number of cores, using efficient partitioning of tasks to threads and cores.

5.9 Conclusion

In this work, We pursue to make strong use of parallelization in order to achieve the required high throughput in packet classification. We have designed, implemented and evaluated Pnpf, a parallel traffic aware packet classification system that exploit the locality in traffic patterns and the task-level parallelism on multi-core general purpose processors to achieve Gbps classification

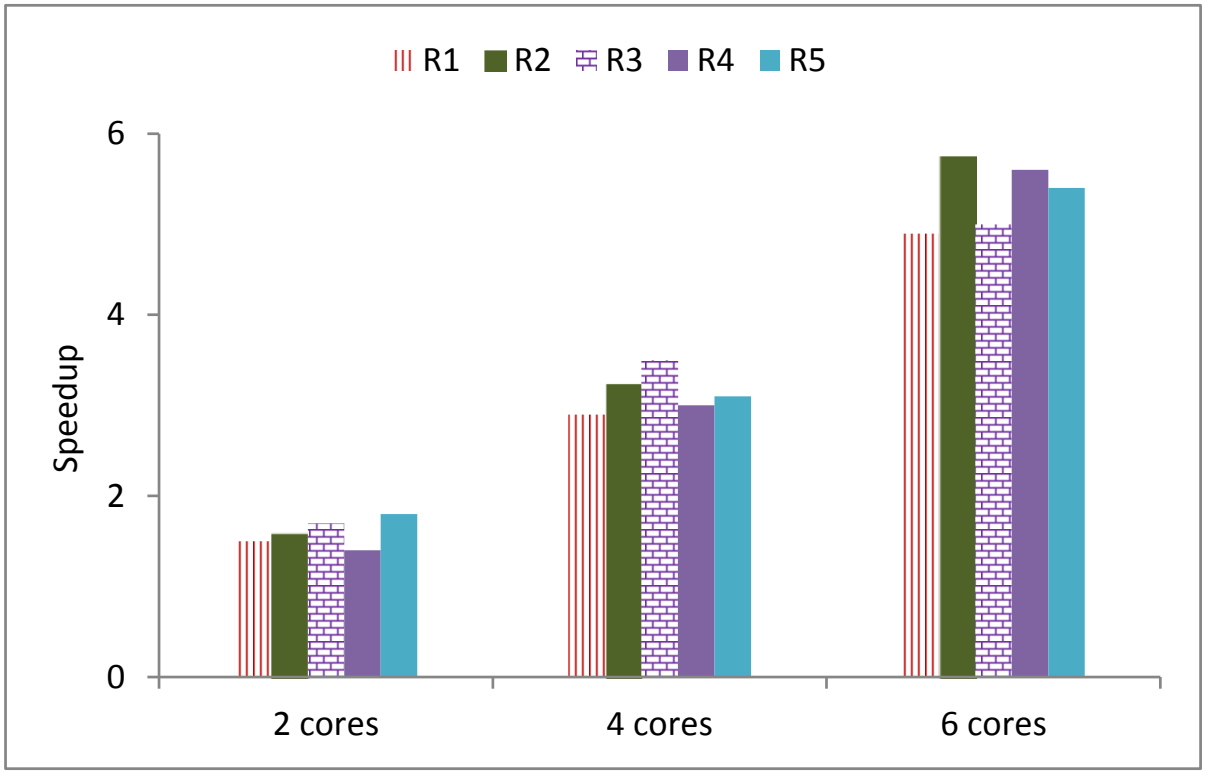


Figure 5.11: `Pnpf`'s scalability with number of cores

speed. We utilize pipeline model (one type task parallelism) to parallelize `npf` because of its inherent modular nature and the benefits of pipeline model such as reduced lock contention, reduced latency, and good cache locality compared to other types of parallelism, ex. data parallelism. However, one limitation of pipeline model is that applications parallelized using this model are very sensitive to the load balancing across the stages. Our design includes controller that dynamically balances load across the stages to ensure an overall good throughput. Experimental results show that by using a 6-core AMD Opteron processor on desktop platform, it is possible to sustain classification rates of more than 10 Gbps for a 5-dimensional rule-set.

BIBLIOGRAPHY

[1] Adel El-Atawy, Taghrid Samak, Ehab Al-Shaer, and Hong Li. Using online traffic statistical matching for optimizing packet filtering performance. In *Proceedings of INFOCOM*, 2007.

[2] Hazem Hamed, Adel El-Atawy, and Ehab Al-Shaer. On dynamic optimization of packet matching in high-speed firewalls. *IEEE Journal on Selected Areas in Communications*, 24(10):1817–1830, 2006.

[3] *National Laboratory for Applied Network Research - Passive Measurement and Analysis*, December 2003. Available as `http://pma.nlanr.net/Special/auck8.html`.

[4] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of ACM SIGCOMM 2003*, August 2003.

[5] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[6] David E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys*, 37(3):238–275, September 2005.

[7] George Varghese. *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[8] *IDT Network Search Engines*. Available in `http://www.idt.com/products`.

[9] Anthony J. Mcauley and Paul Francis. Fast routing table lookup using cams. In *IEEE INFOCOM*, pages 1382–1391, 1993.

[10] Qunfeng Dong, Suman Banerjee, Jia Wang, and Dheeraj Agrawal. Wire speed packet classification without tcams: a few more registers (and a bit of logic) are enough. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 253–264, New York, NY, USA, 2007. ACM.

[11] Mark H. Overmars and A. Frank van der Stappen. Range searching and point location among fat objects. *Journal of Algorithms*, 21:629–656, 1994.

[12] Pankaj Gupta and Nick Mckeown. Packet classification using hierarchical intelligent cuttings. In *Proceedings of HOT Interconnects VII*, August 1999.

[13] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '98, pages 191–202, New York, NY, USA, 1998. ACM.

[14] Thomas Y. C. Woo. A modular approach to packet classification: Algorithms and results. In *In IEEE INFOCOM*, pages 1213–1222, 2000.

[15] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of ACM SIGCOMM*, 1998.

[16] Florin Baboescu and George Varghese. Scalable packet classification. In *Proceedings of ACM SIGCOMM 2001*, August 2001.

[17] David E. Taylor and Jonathan S. Turner. Scalable packet classification using distributed crossproducing of field labels. In *INFOCOM*, pages 269–280, 2005.

[18] Anja Feldmann and S. Muthukrishnan. Tradeoffs for packet classification, 2000.

[19] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proc. of SIGCOMM*, pages 135–146, 1999.

[20] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *Proceedings of ACM SIGCOMM '99*, 1999.

[21] Edith Cohen and Carsten Lund. Packet classification in large isps: design and evaluation of decision tree classifiers. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 73–84, New York, NY, USA, 2005. ACM.

[22] Kun-Chan Lan and John S. Heidemann. A measurement study of correlations of internet flow characteristics. *Computer Networks*, 50(1):46–62, 2006.

[23] Kai Zheng, Chengchen Hu, Hongbin Lu, and Bin Liu. A TCAM-based distributed parallel IP lookup scheme and performance analysis. *IEEE/ACM Trans. Netw*, 14(4):863–875, 2006.

[24] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. *SIGPLAN Not.*, 34:1–12, May 1999.

[25] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4 edition, 2006.

[26] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[27] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[28] Anderson Bailey. *Using Threaded Queues to Get Around Slow I/O*. Available as `http://developer.amd.com/Pages/1024200676.aspx`.

[29] *Multi-threaded programming: efficiency of locking*. Available as `http://attractivechaos.wordpress.com/2011/10/06/multi-threaded-programming-efficiency-of-locking/`.

[30] Randy Smith, Dan Gibson, and Shijin Kong. To cmp or not to cmp: analyzing packet classification on modern and traditional parallel architectures. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, pages 43–44, 2007.

[31] *Built-in functions for atomic memory access*. Available as `http://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html`.

[32] Alexander Sandler. *Multithreaded simple data type access and atomic variables*. Available as `http://www.alexonlinux.com/multithreaded-simple-data-type-access-and-atomic-variables#how_atomic_variables_work`.

[33] Frank Mueller. Pthreads library interface. *Technical report, Department of Computer Science, Florida State University*, July 1995.

[34] David E. Taylor and Jonathan S. Turner. Classbench: A packet classification benchmark. In *Proceedings of IEEE INFOCOM*, 2004.

[35] A. Foong, J. Fung, and D. Newell. An in-depth analysis of the impact of processor affinity on network performance. In *IEEE ICON*, 2004.

[36] Shariful Hasan Shaikot and Min Sik Kim. npf—a simple, traffic-adaptive packet classifier using on-line reorganization of rule trees. In *Proceedings of the 5th IEEE LCN Workshop on Security in Communications Networks*, October 2009.

[37] Lukas Kencl and Christian Schwarzer. Traffic-adaptive packet filtering of denial of service attacks. In *Proceedings of the 2006 IEEE WoWMoM*, June 2006.

[38] P. Gupta, B. Prabhakar, and S. Boyd. Near-optimal routing lookups with bounded worst case performance. In *Proceedings of IEEE INFOCOM '00*, March 2000.

[39] H. Hamed, A. El-Atawy, and E. Al-Shaer. Adaptive statistical optimization techniques for firewall packet filtering. In *Proceedings of IEEE INFOCOM*, April 2006.

[40] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill, 1990.

[41] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[42] *Snort User Manual 2.8.3*. Available as `http://www.snort.org/assets/125/snort_manual-2_8_5_1.pdf`.

[43] The NetBSD Foundation. *NetBSD Kernel Developer's Manual*, April 2008. Available as `http://man.NetBSD.org/man/+ANY+NetBSD-5.0`.

[44] The NetBSD operating system. `http://www.netbsd.org/`.

[45] `tcpreplay`. `http://www.tcpreplay.synfin.net`.

[46] *Removing System Bottlenecks in Multi-threaded Applications*. Available as `download.intel.com/design/intarch/papers/320631.pdf`.

[47] *Supra-linear Packet Processing Performance with Intel Multi-core Processors*, 2006. Available as `download.intel.com/technology/advanced_comm/31156601.pdf`.

[48] Francesco Fusco and Luca Deri. High speed network traffic analysis with commodity multi-core systems. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pages 218–224, New York, NY, USA, 2010. ACM.

[49] C. Networks. *Octeon network service processors*. Available as `http://www.cavium.com/octeon_software_develop_kit.html`.

[50] JK. Lan and J. Heidemann. On the correlation of internet flow characteristics. Technical report, Technical Report ISI-TR-574, USC/ISI, 2003.

[51] L A Barroso, K Gharachorloo, R McNamara, A Nowatzyk, S Qadeer, B Sano, S Smith, R Stets, and B Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. *Proceedings of 27th International Symposium on Computer Architecture IEEE Cat NoRS00201*, 28(2):282–293, 2000.

[52] P Kongetira, K Aingaran, and K Olukotun. Niagara: a 32-way multithreaded sparc processor, 2005.

[53] Cristian F. Dumitrescu. *Design Patterns for Packet Processing Applications on Multi-core Intel Architecture Processors*, 2008. Available as `download.intel.com/design/intarch/papers/321058.pdf`.

[54] *Quad Core Intel Xeon Processor 5300 Sequence*. Available as `ftp://download.intel.com/products/processor/xeon/dc53kprodbrief.pdf`.

[55] *Six Core Intel Xeon Processor*. Available as `http://ark.intel.com/cpu.aspx?groupID=36937`.

[56] Sun Microsystems. *UltraSPARC T2 Overview*. Available as `http://www.sun.com/processors/UltraSPARC-T2/features.xml`.

[57] Tilera Corporation. *Tilera Tile64 Processor*. Available as `http://www.tilera.com/products/TILE64.php`.

[58] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[59] Inc. Cisco Systems. *Application-Oriented Networking: Products and Services*. Available as `http://www.cisco.com/en/US/products/ps6692/Products_Sub_Category_Home.html`.

[60] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[61] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[62] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 15–28. ACM, 2009.

[63] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[64] Srinivas N. Vadlamani. The synchronized pipelined parallelism model. In *In The 16 th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2004.

[65] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *In Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, pages 105–118. IEEE Computer Society, 2005.

[66] John Giacomoni and Manish Vachharajani. Fastforward for concurrent threaded pipelines. Technical report, University of Colorado at Boulder, 2007.

[67] Jan Van Lunteren, Ton Engbersen, and Senior Member. Fast and scalable packet classification. *IEEE Journal on Selected Areas in Communications*, 21:560–571, 2003.

[68] Y Qi, L Xu, B Yang, Y Xue, and J Li. Packet classification algorithms: From theory to practice. *IEEE INFOCOM 2009 The 28th Conference on Computer Communications*, pages 648–656, 2009.

[69] Haipeng Cheng, Zheng Chen, Bei Hua, and Xinan Tang. Scalable packet classification using interpreting: a cross-platform multi-core solution. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 33–42, New York, NY, USA, 2008. ACM.

[70] Danhua Guo, Guangdeng Liao, Laxmi N. Bhuyan, Bin Liu, and Jianxun Jason Ding. A scalable multithreaded l7-filter design for multi-core servers. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 60–68, 2008.

[71] Kai Zheng, Zhiyong Liang, and Yi Ge. Parallel packet classification via policy table prepartitioning. In *Proceedings of Global Telecommunications Conference 2006, dec 2005*, GLOBECOM 05, 2005.