ClassiPI: An Architecture for Fast and Flexible Packet Classification

Sundar Iyer, Ramana Rao Kompella, and Ajit Shelat
PMC-Sierra Inc.

Abstract

Packet classification is a fundamental and critical operation to be performed in networking equipment such as switches and routers. The types of classification to be performed encompass a wide range, from well-understood operations such as route table lookups to complex packet identification involving multiple fields in the packet. Furthermore, the advent of application-aware network devices demand the use of flexible packet classifiers that can handle operations such as pattern searches and regular expression matching. Traditionally, depending on the classification types to be implemented, solutions based on CAMs/TCAMs, special function ASICs, and software-based algorithms have been used. These solutions, while adequate, target specific classification applications. This article describes ClassiPI™, a programmable hardware architecture that performs packet classification at OC48c line rates. Illustrations and examples show how this architecture can be used in conjunction with software algorithmic techniques to support the classification needs of a variety of applications from packet switching, forwarding, and filtering to layer 7 applications such as server load balancing. We believe that the ClassiPI architecture is scalable and flexible to meet the needs of a broad class of network applications.

Network equipment today need to perform complex packet analysis as part of their packet processing. The performance of the packet analysis or classification module directly affects the overall performance of the equipment. This becomes particularly critical for line rates starting from Gigabit Ethernet (GE) and going up to OC192c. Classification requirements also vary with the applications and the equipment categories for which they are intended.

To date, to meet these needs, special-purpose classification devices, such as CAMs/ternary CAMs (TCAMs), special route lookup engines, and software-based classification algorithms have been deployed. However, they target specific applications. Since, at this stage of evolution of the Internet, network applications (and hence classification requirements) continue to evolve, there is now a definite need for a classification architecture that is flexible enough to support a variety of applications at wire speeds. The ClassiPI architecture defines a programmable, high-speed classification engine that attempts to address this need. The first implementation of the ClassiPI architecture achieves wire-speed classification at OC48c rates.

ClassiPI as a co-processor is intended to be used as an accelerator on line cards and in systems based on application-specific integrated circuits (ASICs), network processors, or general-purpose processors. It is deliberately agnostic with respect to packet processors and does not impose any constraints on them. It can accept a stream of data from the processor, perform classification operations on this stream, and return classification results back to the processor, as shown in Fig. 1.

The rest of this article is organized as follows. We briefly present an overview of classification. This is followed by a summary of existing solutions. We then introduce various performance metrics to help characterize classification. An abstraction of classification is presented, and the ClassiPI architecture is described. Examples of some applications that can be supported on ClassiPI are described, followed by performance figures.

A Classification Overview

Given a set of rules or policies defining packet attributes or content, packet classification is the process of identifying the rule or rules within this set to which a packet conforms or matches.

Rules typically consist of operations comparing packet fields with values, as shown in Table 1. A set of rules is formed based on the criteria to be applied to classify packets with respect to a given network application. Classification rule sets can be categorized based on the application as described below:

- **Packet forwarding applications**: Examples of these include MAC address-based layer 2 (L2) switching, asynchronous transfer mode (ATM) cell switching, multiprotocol label switching (MPLS), and layer 3 (L3) IP forwarding, as shown in Table 1. The classification operation required in these applications is usually performed on a single field in the packet header. For example, L3 IP forwarding implements a longest prefix match policy on the destination IP address of the packet.
- **Packet-filter-based applications**: Examples of these are firewall packet filtering, virtual private network (VPN) implementations, and quality of service (QoS) applications such as integrated services (IntServ) and differentiated services (DiffServ). Typically, these applications use policies based on L2/L3/L4 fields in the packet header. These applications

Layer	Application	Speeds	Number of fields	Classification type	Rule example
Two	Switching, MPLS	OC48c	Single	Exact match	Send packets with DMAC == 68:10:01:ab:12:7a directly to end hosts
Three	Forwarding	OC48c	Single	Longest prefix match	Send all packets with DIP == 192.168.0.* to the ISP,s router
Four	Flow identification, IntServ	OC48c	Multiple	Exact match	Give packet with SIP, DIP, SP, DP == (192.1.4.5, 200.10.2.3, 21, 1030) highest priority
Four	Filtering, DiffServ	OC48c	Multiple	Prefix or range match	Drop all packets with SIP == 192.1.* and SP > 1023 and DP < 5000
Seven	Load balancing	1GE	Multiple	Scan with exact or prefix match	Re-direct packets having filenames ending in ".ra" in DATA to audio server.
Seven	Intrusion detection	1GE	Multiple	Scan and match reg. expressions	Create alarm when packets having "get.*vbs" in DATA arrive.

■ Table 1. *Packet classification examples.*

have rules that require an exact match or prefix/range match on multiple fields, as shown in Table 1.

- **Content-aware applications**: These are a new class of applications requiring scanning and classification based on the packet header as well as the packet content. Examples are server load balancing, intrusion detection, and virus scanning (Table 1). Classifiers for these applications require scanning the packet to locate specific fields.

From the above it is evident that classification requirements vary with the application.

Existing Approaches

Several approaches and algorithms have been proposed to solve the problem of packet classification. Some of them are discussed in other tutorial articles in this special issue. In this section we will provide a short summary of some of the existing approaches.

A brute force approach to classification is to precompute the rule that matches for every possible packet header. Classifiers that act on data of width w would require a table of size 2^w entries. The classification result is obtained in a single memory access; however, it requires an exponentially large amount of memory to be practical.

The simplest classification algorithm is a linear search that involves comparing each rule sequentially with the incoming data until a match is found. Though the memory for rule storage is used efficiently, the search time scales linearly with the number of rules, making it impractical for use with large rule databases.

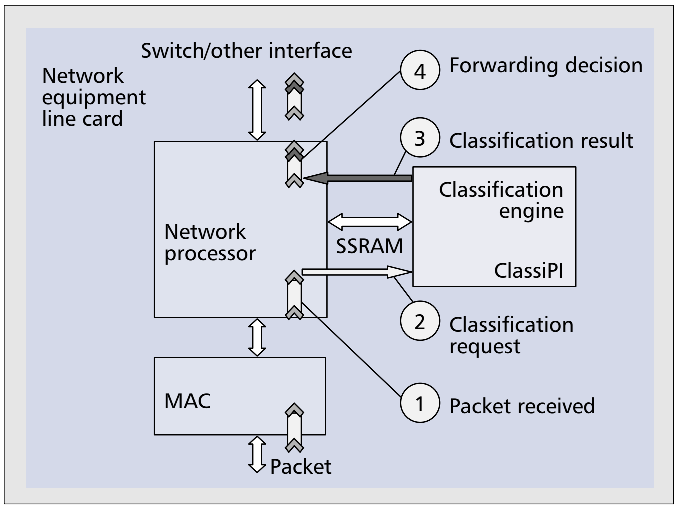
CAMs and TCAMs [1, 2] perform classification by matching the incoming data against all rules in parallel and reporting the highest priority match. This requires a large number of logic elements. Hence they suffer from power dissipation problems and are not scalable to accommodate large rule databases. Also, in a TCAM, the classification rules can only match a 0, 1, * (i.e., a don,t care), for each of the bit positions of the incoming data. This can be restrictive for certain kinds of classification applications.

IPv4 packet forwarding is a special case of packet classification that involves single field classifiers. Various algorithms that pre-process the rule database and organize it as a tree/trie have been proposed in [3–6]. In [7], the authors propose a two-level table lookup mechanism to perform packet forwarding. The above algorithms are optimized only for IPv4 forwarding and do not address the general packet classification problem.

Packet-filtering algorithms based on multiple fields have been studied in [8–15]. These algorithms maintain a multidimensional tree or trie data structure for the rule database. Though efficient, they cater only to the specific filtering-based applications and are not suitable for applications requiring content-aware classification.

A requirement of content-aware classification is the need to scan or parse [16] a packet and check for the occurrence of one or more strings or patterns defined in a wide-width rule database.

In the next section we define metrics which can be used to compare the performance of different classification solutions.



■ Figure 1. *ClassiPI as a co-processor.*

Classification Performance Metrics

Definition 1: Space Complexity (S) — *This is defined as the asymptotic tight bound* $\Theta(f(n))$[1] *on the space required to represent a particular rule database and its associated data structures.*

For example, a linear search software algorithm has a space complexity of $\Theta(n)$. A TCAM also has a space complexity of $\Theta(n)$. However, the actual space required is $c \geq 1$, for some c depending on the complexity of the rule.[2]

[1] *In the rest of the article,* n *refers to the number of rules.*

[2] *As an example, a rule of the form* x > *1023, for some field* x*, requires six TCAM rules to store. Hence c may be as large as 6.*

Algorithms/architectures	S	T	P	R
Table	High	Low	Low	Low
Linear	Low	High	High	High
TCAM	Low	Low	High	Medium
Single-dimension trie/trees	Medium	Medium	Low	Low
Multidimension trie/trees	High	Partially	Low	Low
Ideal	Low	Low	Low	High

Table 2. *Packet classification trade-offs.*

Definition 2: Time Complexity (T) — *This is defined as the asymptotic tight bound $\Theta(f(n))$ on the maximum number of steps or cycles that are required to perform the classification.*

This metric is used to identify the number of sequential steps required to reach a match/no match result. Any parallelism implemented helps to reduce T. Algorithms that narrow down the number of rules to match (and hence the number of steps) would also reduce T.

For example, a time-wise linear search algorithm has a time complexity of $T = \Theta(n)$. A standard binary tree search algorithm has a time complexity of $T = \Theta(\log n)$ for certain single-field classifiers. CAMs, being completely parallel implementations, have $T = \Theta(1)$.

Definition 3: Power Complexity (P)[3] — *This is defined as the asymptotic tight bound $\Theta(f(n))$ on the product of the maximum number of steps or cycles that are required to perform classification, the number of operations that are performed in parallel per step, and the cost, in terms of power required, of a single operation.*

This metric attempts to identify the number of basic match operations performed and, by implication, the power required. Thus, a linear lookup, whether time-wise or space-wise, would perform the same number of lookups and hence have the same power complexity.

For example, a linear search algorithm has a power complexity of $P = \Theta(n)$. A binary tree search algorithm (for certain single field classifiers) has a power complexity of $P = \Theta(\log n)$. A direct table lookup has $P = \Theta(1)$. A hash table lookup with bucket size b, has $P = \Theta(b)$.

Definition 4: Update Complexity (U) — *This is defined as the asymptotic tight bound $\Theta(f(n))$ on the maximum number of steps or cycles required to perform an atomic insert, delete or update of a rule in the rule database.*

For example, a linear search algorithm requires an update time of $U = \Theta(n)$. Also, most CAMs require $U = \Theta(n)$ for L3/L4 access control lists, however certain specialized CAM engines [1] require $U = \Theta(1)$ update time. A lower update complexity $U = \Theta(w)$ for CAMs, specifically for L3 lookups has been shown in [17]. In [9], Buddhikot *et al.* report an update complexity of $U = \Theta(\wp n)$. Also, a complexity of $\Theta(ln^{1/l})$ to insert $\Theta(n^{1/l})$ rules is derived in [10].

Although an important metric for brevity, it is not discussed further in this article.[4]

[3] *Although power complexity can be more precisely defined, we assume a simplistic definition for this metric.*

[4] *However, we would like to mention here that the update complexity for algorithms implemented on the ClassiPI architecture depends on the application and the algorithm used for classification, and considerable flexibility has been provided in the architecture to fine tune this parameter.*

Definition 5: Rule Complexity (R) — *This is defined as the product of the number of operations which can be supported per field of the rule and the total number of fields in the rule.*

This metric reflects the complexity of the policy rules it can support. It can be used to estimate the range of applications that the engine can support. Examples of such operators include comparison operators, packet parsing operators, concatenation of fields to form variable width fields, and a wide range of regular expression operators.

For example, a unary CAM has only a single comparison operator (i.e., the equality match operation). Hence, its rule complexity is $1 \times F$, where F is the number of fields. A TCAM, on the other hand, has $R = 2 \times F$ since it can perform both an equality and a masking operation per field. A solution which implements additional relational operators such as $<,>$ in addition to TCAM operators has a rule complexity $R = 4 \times F$.

A Classification Wish List

The following is a possible wish list for an optimally architected classification engine:

- It should be efficient and scalable in the usage of memory (i.e., have a low *space complexity*, S).
- It should have a high classification speed (i.e., have a low *time complexity* T).
- It should consume less power (i.e., have a low *power complexity*, P), enabling the solution to be scalable.
- The architecture should also be able to support fast incremental inserts and deletes of the rules in the rule database (i.e., have a low *update complexity* U).
- The architecture should have a high *rule complexity* R.

The classification trade-offs for the solutions described in the previous section are illustrated in Table 2. Also, from a flexibility and application point of view we would additionally require the following:

- It should support multiple different classification algorithms so that the optimal algorithm may be implemented for a particular application. For example, a linear search algorithm for a small number of rules may perform better than a tree/trie search algorithm.
- Typically, applications require multiple classifications, and hence the architecture should be able to perform a sequence of classification operations on a single packet. An example of such a sequence is explained later.

Theory suggests that packet classification has inherent trade-offs between the various metrics defined above. Table 2 shows that existing solutions are inefficient with respect to at least one metric.

Table 3 illustrates the theoretical complexity metrics of ClassiPI with respect to rule sets for IPv4 route lookup.[5] In the table, w refers to the width of the field used for classification in bits. The complexity metrics of ClassiPI have been

Algorithm/architecture	*S*	*T*	*P*
Table	2^w	1	1
Linear	n	n	n
CAM	n	1	n
Balanced binary search	$4n - 1$	$\log_2 2n$	$\log_2 2n$
ClassiPI using B-Tree	$2n + (2n - p)/(p - 1)$	$\log_p 2n$	$(p - 1)\log_p 2n$
Ideal	n	1	1

Table 3. *Trade-offs for IPv4 route lookup.*

obtained using an order p B-Tree algorithm with p comparisons in parallel. The tree is formed on a worst case number of $2n$ disjoint intervals[6] which are formed from n routing prefixes. Each of these intervals is associated with a best matching prefix. The exact algorithm is explained later. The balanced binary search algorithm is a special case of this, where the B-Tree has an order of $p = 2$. The search complexity of the balanced binary search tree is $\log_2 2n$ and that of a order p B-Tree is $\log_p 2n$. The power complexity of an order p B-Tree is the product of the number of elements in the internal nodes and the depth of the tree i.e. $(p - 1)\log_p 2n$.

We see that ClassiPI, when implementing a B-Tree algorithm, has a moderate S, T, P complexity for corresponding values of p; the internal parallelism of ClassiPI. Implementations of the classification algorithm on the ClassiPI architecture can be tuned to optimize a particular metric by choosing appropriate values of p. In particular, p can be set to 1, mimicking a simple linear search. The next section lays the foundation for describing the ClassiPI architecture.

Operators	Examples
Arithmetic	MASK
Logical	AND, OR, NOT
Relational	<, >, ==
Repetition	*

■ Table 4. *Operators in a rule.*

A Framework for Classification

In this section we describe a framework for packet classification in two parts:
- The primitives that form the building blocks for classification
- The sequencing operators that combine these primitives to provide a complete classification solution

A Formalization of Classification Primitives

Definition 6: Field — *The field* F_i *is a contiguous set of bits that can be either a part of the packet, or other information attached or associated with the packet, such as timestamp or incoming port number.*

The task of packet parsing involves identification of the fields in the packet. This operation is simple in the case of L2/L3/L4 applications where the field offsets are either known with respect to the start of the packet or dependent on the contents of another field. Examples of fields involved in packet classification include SIP, DIP, SP, DP, and IP Protocol.

Definition 7: Field List (Key) — *A field list or key* $F = [F_0, F_1, \ldots, F_n]$ *is an ordered set of fields.*

For example, a key for identifying a unique connection can comprise the ordered list F = [SIP, DIP, SP, DP, IP Protocol].

Definition 8: Rule — *This is defined as a Boolean function,* $\rho : F \rightarrow true, false$ *where* $F = \{F[0], F[1], F[2]..\}$ *is the key or field list. In case the rule matches the fields the function returns true. It returns false otherwise.*

In general, any Boolean function can constitute a rule; however, typical operators used in defining a rule are given in Table 4.

While the L2/L3/L4 header fields are easily parsed, content-aware classification rules often require fields located within the data packet payload. In this case, the offsets of fields may not be known a priori, and the rule must encode the parsing and classification information. The last column of Table 1 shows some examples of rules and the actions associated with them. Recall that the number of fields in the key and the number of operators supported by the architecture define the rule complexity metric R.

Definition 9: Rule List — *A rule list* $\Re$ *is defined as an ordered set of rules* $[\rho_1, \rho_2, \ldots, \rho_n]$.

Definition 10: Selection Operator — *The selection operator is a function* $\sigma: \Re \rightarrow \Re$*, which chooses a subset of rules from a set of rules.*

For example, a highest-priority selection operator, reports only the highest-priority rule among all the rules that matched a key. If the rule list is arranged in increasing order of priority, the highest-priority rule corresponds to the first rule that matched.

Definition 11: Classification Function (CF) — *The classification function* π *is a function on a rule list* $\Re$ *with field set* F *and selection operator* σ*, as* $\pi : (\Re, F, \sigma) \rightarrow \Re$*. The classification function selectively returns the rules which matched field set* F *based on the selection operator* σ*.*

For example, the classification function π in an IPv4 forwarding application is defined by a rule list $\Re$ of routing prefixes, a key F = [DIP], and a selection operator σ which returns the longest matching prefix out of all the prefixes that matched. The various approaches, some of which have been discussed, implement the classification function in different ways.

Definition 12: Boolean Match Operator — *The Boolean match operator: match index* $\pi(\Re, F, \sigma) \rightarrow true, false$ *returns true if* $\exists_i(\rho_i\ (\Re, F, \sigma))$ *and false otherwise.*

In other words, the Boolean match operator returns a value *true* if there is at least one rule in the rule set that matches the field list F.

Definition 13: Boolean Rule Match Operator — *The Boolean rule match operator: match index* $\pi(\Re, F, \sigma)$*,* $i \varnothing$ *true, false returns true if* $\rho_i \in \pi(\Re, F, \sigma)$ *and false otherwise.*

The Boolean rule match operator compares the return value of a classification function with a rule index to check if they match. Both the above Boolean operators, when invoked, first perform the classification function π and then check for the Boolean condition.

The above definitions are fairly generic and can be mapped onto classification devices such as CAMs, TCAMs, and route lookup engines. For example, each entry in a CAM corresponds to a rule ρ. The only Boolean function supported by a rule is the equality operator '=='. The rule list consists of all the entries in a CAM. The classification function π consists of matching the key or field list against all the entries in the CAM. The Boolean rule match operator returns the index/address of the highest-priority entry which matches the key. In the case of a CAM, the various metrics are $S = \Theta(n)$, $T = \Theta(1)$, $P = \Theta(n)$.

A Formalization of Sequencing Operators

The following definitions lay the framework to combine the above-defined classification primitives.

Definition 14: Classification Sequence Descriptor — *A classification sequence descriptor* Π *is a series of classification operations that have an order of execution associated with them. This is illustrated as follows, where,* $\Pi_1 \bullet \Pi_2$ *and* $(Bool)?\Pi_1{:}\Pi_2$ *are the sequencing and conditional sequencing primitive, which are defined below.*

[5] *We do not mention the rule complexity in the table because it is a property of the architecture and is not traded off with the other parameters.*

[6] *Strictly speaking, the worst case number of disjoint intervals is 2n + 1. We use 2n to keep the calculations simple.*

$$\Pi = \Pi_1 \bullet \Pi_2 \big| (Bool)?\Pi_1 : \Pi_2 \big| \pi \,|\, \phi$$

$$Bool = match(\pi) \,|\, matchindex(\pi, i)$$

$$i = \text{constant}$$

Definition 15: Sequencing Operator — ***The sequencing operator** $\Pi_1 \bullet \Pi_2$ **implies the execution of** Π_2 **is after the execution of** Π_1.*

When two classification sequence descriptors Π_1 and Π_2 are executed in sequence, the metrics are calculated as follows:

$$S = S(\Pi_1) + S(\Pi_2)$$
$$T = T(\Pi_1) + T(\Pi_2)$$
$$P = P(\Pi_1) + P(\Pi_2)$$

Definition 16: Conditional Sequencing Operator — *This operator is denoted by (Bool)? Π_1: Π_2. It implies that if Bool is true, then Π_1 is executed, else Π_2 is executed. It can also be denoted as if (Bool) then Π_1 else Π_2.*

When two classification sequence descriptors Π_1 and Π_2 are conditionally executed, the metric values are calculated as given below.

$$S = S(Bool) + S(\Pi_1) + S(\Pi_2)$$
$$T = T(Bool) + \max\{T(\Pi_1) + T(\Pi_2)\}$$
$$P = P(Bool) + \max\{P(\Pi_1) + P(\Pi_2)\}$$

Here, the metric associated with the *Bool* is the cost of the π operation associated with *Bool*.

Definition 17: Switch-Case Operator — *The switch-case operator is used conditionally to select the next operation to be performed. It is represented as switch($\pi(\Re$, F, P)): (Π_1, Π_2, Π_3, … Π_n), and implies that Π_i is executed if matchIndex($\pi(\Re$, F, P), i) is true.*

The switch-case operator can be derived using multiple conditional sequencing operators and is hence not a primitive operator. When a switch-case operator is executed, the metric values are calculated as follows:

$$S = S(Bool) + \sum_{i=1}^{n} \{S(\Pi_i)\}$$
$$T = T(Bool) + \max\{T(\Pi_1), T(\Pi_2), \ldots T(\Pi_n)\}$$
$$P = P(Bool) + \max\{P(\Pi_1), P(\Pi_2), \ldots P(\Pi_n)\}$$

Using these primitives and operators, the classification engine can be used to implement lookup algorithms that optimize the metrics *S*, *T*, *P* for a particular application.

Later, we shall illustrate the implementation of two applications with the help of primitives defined above.

ClassiPI Architecture

This section briefly describes the ClassiPI architecture and how its various functional units relate to the above definitions. Figure 2 shows a block diagram indicating the main functional blocks of the device.

The classification engine (CE) block, assisted by the control/sequencer block, implements the core classification capabilities of ClassiPI. The other blocks provide the functionality required to interface ClassiPI to other devices in the system, extract the payload, and present the desired commands and parameters to the CE. More information about important blocks is provided in the following sections.

ClassiPI is efficiently pipelined, enabling a continuous stream of packets to be fed into the device, while it continues to perform packet parsing, key formation, and lookup operations.

System Interface

The ClassiPI co-processor presents a general-purpose synchronous random access memory (SRAM) (SyncBurst or ZBT mode) bit interface for connecting to a processor, packet source, or DMA device. The system interface (SI) is used to send packets or pre-extracted payload for classification. The interface supports up to independent channels, each of which appears to be an independent classification engine having a separate area in the *packet buffer* in which associated packet data can be stored. This interface is also used to access the control registers with the help of which classification operations, key selections, and a rule database can be configured. The results of a classification operation are also returned to the processor via this interface.

Field Extraction Engine

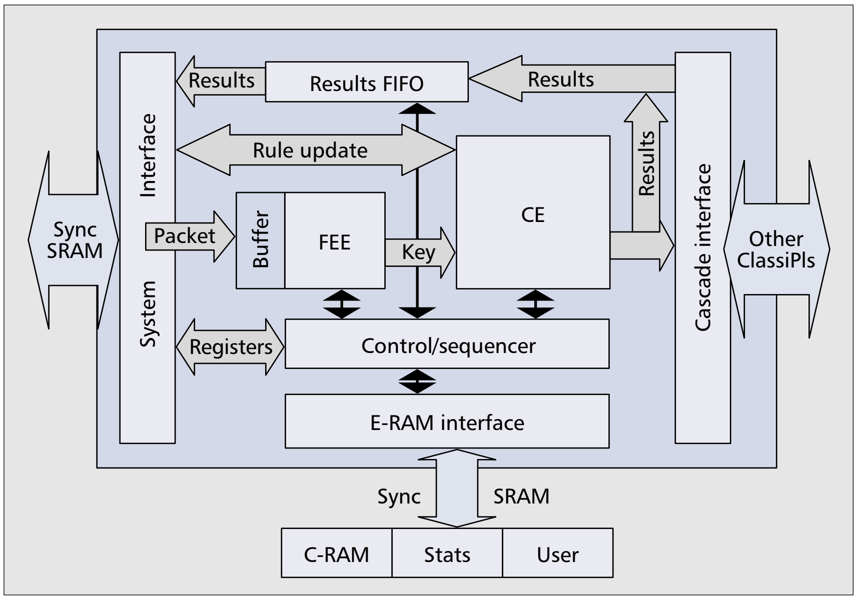
The main function of the field extraction engine (FEE) is to form the key or field list *F* required for a classification operation π. The FEE can parse and extract Ethernet II, 802.3, and 802.1*p*/*q* headers, then determine where in the packet the L3 payload starts. It can extract key IP, TCP, and UDP header fields as well as payload data at any offset. The FEE can also extract some amount of TCP state information. The FEE is aware of and can handle the various idiosyncrasies of the IP and TCP headers. In addition to performing fixed header extraction, the FEE can also be programmed to extract data between a start and an end offset in the input data stream. These offsets can be either preprogrammed using the control registers, or dynamically obtained from the results of the previous classification operation. It is also possible to bypass the FEE on a per-packet basis and send a pre-extracted key directly to ClassiPI.

Classification Engine

The classification engine (CE), implements a set of classification functions (CFs). Each CF implements a π operation and can be programmed to be associated with a rule set $\Re$, a key *F*, and a selection operator σ. The rule set $\Re$ is allocated space from a common pool of 16K rules. This rule space can be extended to 128K rules using a cascade interface, as explained later.

Rule Capabilities — The architecture supports a wide range of rule formats. The format allows specification of ranges using the >,<, and MASK operators. It also supports the scan operator, *, and other operators listed in Table 4. This degree of flexibility built into the rule format ensures a high value of rule complexity *R* for ClassiPI. The architecture allows the CE to be configured to create multiple CFs each with varying number of rules, rules of varying complexity and keys of varying widths. The rules are configurable in a variety of ways, from 108-bit L4 classifiers to wide-width classifiers of up to 192 bytes for L7 applications.

Search Operation — The fundamental operation on the CE is the search operation π. The search operation selects a particular and proceeds by performing a match operation for each rule in the CF. The results depend on the selection operator specified in the search operation. The ClassiPI architecture supports two kinds of selection operators, Highest Priority Match and Multiple Match. The Highest Priority Match operator returns the rule which has the smallest index out of all the rules that match the packet data. In the case of a Multiple Match, all the rules that match the packet data are reported. The results of a Multiple Match search operation are queued



■ Figure 2. *A ClassiPI block diagram.*

up in the Results FIFO (Fig. 2). The processor can access these results and abort this operation at any stage. Thus, any of the top *m* out of *n* rules that matched can be reported ($m \le n$).

The Highest Priority Match operation is used in applications such as L4 filtering and DiffServ. The Multiple Match search operation could be used to implement applications such as RMON, which maintain statistics such as number of IP packets, TCP packets, and IP fragments, and where more than one rule can match a packet.

External RAM Interface

The co-processor has an interface to an external RAM (ERAM) which is used to store:
- The classification program Π, in the command RAM (C-RAM) section.
- The user programmable data associated with every rule in the user section.
- Certain statistics which are maintained by the device on a per rule basis in the "*Stats*" section. These include packet count, byte count, and timestamp.

ClassiPI Control and Sequencer Block

The ClassiPI control and sequencer block orchestrates the operation of the various blocks to perform packet classification. The ClassiPI control block can be directly fed a classification program Π by the processor or programmed to select Π from the ERAM. The sequencer understands and creates the multiple sequences of classification operations π, as well as the conditional and unconditional search operations in Π. It is responsible for feeding the field list *F* from the FEE to the CE and the selecting the matching rule(s) based on the selection operator σ.

Cascade Interface

The cascade interface can be used to connect up to a maximum of eight ClassiPI devices. The cascade mechanism can be used to increase the number of rules in the common pool to a maximum of 128K. In the cascade configuration, each of the chips receives the same key and operates on different rules in parallel. A large filter database can be distributed among multiple chips and searched in parallel.

ClassiPI Implementation

ClassiPI is available in a 352 pin BGA package and has an estimated power consumption of 2.25 W. It is implemented using 0.18 μ technology. The classification core runs off a 200 MHz clock while the interfaces of the device can be configured to run at 66 or 100 MHz.

Application Examples

Having described the ClassiPI architecture, examples of illustrative applications based on ClassiPI can now be given. The examples selected here are a simple linear lookup, an L3 IP forwarding application, and an L7 server load balancing application. A brief description of the algorithm and implementation strategy is followed by pseudo-code describing the algorithm. The pseudo-code is realized in ClassiPI as follows:
- The classification function π, the key *F*, and the selection operator σ are configured by programming the control registers.
- The corresponding rule set $\Re$ is programmed into the CE.
- Both the conditional and unconditional classification sequence descriptors are specified in the C-RAM.

Linear Search

The linear search mechanism is the simplest method to perform L2, L3, and L4 lookups. One way to do this is by creating a prioritized rule set $\Re$ such that if $\rho_i, \rho_j \in \Re$ and ρ_i has a higher priority than ρ_j, then $i < j$. A field list *F* is then defined and a selection operator σ, selects the highest priority match is chosen. The pseudo-code for the linear search is then simply $\pi(\Re, F, \sigma)$.

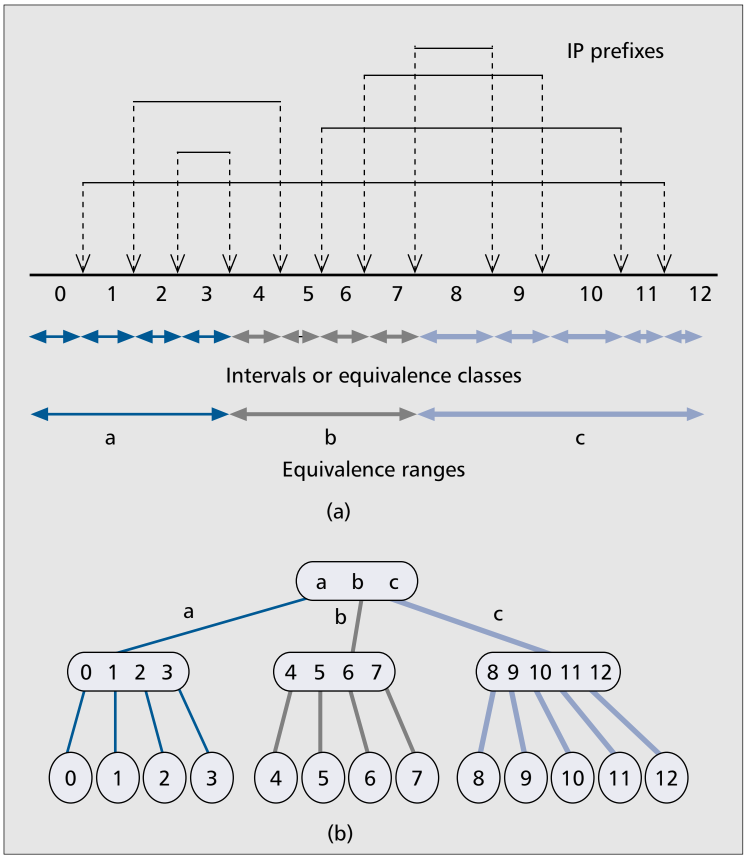
Tree Based IPv4 Routing

This subsection describes the IPv4 forwarding algorithm using one specific algorithm called Interval Search.

Construction Step — The interval search algorithm for IPv4 forwarding first breaks up the IP prefixes (which may overlap) into lower and upper endpoints. Then it sorts all of these endpoints, in increasing order, into one single list of numbers. Each of the two adjacent numbers now form an interval. There can be at most $2n + 1$ such intervals, where *n* is the number of prefixes. These intervals are all disjoint and form an equivalence class [14] since all the points in this interval have the same longest matching prefix. In Fig. 3 there are prefixes represented by ranges, and 13 equivalence classes are formed (0–12 denoted in the figure). To search for a given destination IP address for the longest matching prefix, it is sufficient to now match the IP address in these equivalence classes.

A small number of contiguous equivalence classes are merged to form an encapsulating range. For example, in the figure shown above, the intervals 0, 1, 2, 3 are combined to form encapsulating range *a*, with the lower limit of this range as the lower limit of equivalence class 0 and the upper limit as the upper limit of equivalence class 3. If the destination address of the packet matches encapsulating range *a*, it will match one of the equivalence classes in it. Similarly, encapsulating range *b* has equivalence classes 4, 5, 6, 7, and encapsulating range *c* has equivalence classes 8, 9, 10, 11, 12.

The algorithm builds a multilevel tree on the set of equivalence classes. We shall only describe the construction of such a two-level tree. Rules which consist of encapsulating ranges constitute the root of the tree. This is defined by the root rule list $\Re_0$ Each of the child nodes of this root corresponds to a set of equiva-



■ Figure 3. *a) IP prefixes as disjoint intervals; b) interval tree.*

lence classes. These are represented by rule lists $\Re_1$, $\Re_2$, etc. The field set F consists of the destination IP address. There is a single selection operator which is the identity function (i.e., it returns all the rules that matched). Note, the rule database is constructed so that at any given time, only one interval will match.

Classification Steps — The search takes place in two steps. In the first phase, the root partition $\Re_0$ is searched for a match. Depending on which entry matched in the root, a different set of equivalence classes $\Re_1$, $\Re_2$, and so on would be searched for a match. For example, if there are exactly 10,000 intervals, they could be split up into two levels with 100 encapsulating range entries in the first level and 100 entries in each of the rule lists $\Re_1$, $\Re_2$ … $\Re_{100}$, in the second level.

Based on the theoretical framework laid in previous sections, the pseudocode for implementing the search is as follows:

```
/* Definition */
/* Defining the equivalence classes */
$\Pi_1 = \pi(\Re_1, F, \sigma)$,
$\Pi_2 = \pi(\Re_2, F, \sigma)$,
…,
$\Pi_n = \pi(\Re_n, F, \sigma)$,
/* Execution */
/* Search on the Equivalence Ranges */
switch($\pi(\Re_0, F, \sigma)$):($\Pi_1, \Pi_2, \dots, \Pi_n$)
```

The S, T, P values can be derived directly from the metrics of the switch-case statement defined in the previous section.

Server Load Balancing

We now look at an L7 load balancing application where it is necessary to parse and classify the URL string contained in a packet. Figure 4 shows the various classification operations required; they involve the following stages:

- Check that the *destination port number* field is 80.
- Search for the *host name* field in the packet.
- Do an exact match on the *host name* field.
- Search for the *URL* field in the packet.
- Do a longest prefix match on the URL field.

Construction of the Solution — These five steps are further described below:

1 A rule space $\Re_1$ is defined with a single rule ρ_1, which matches the destination port number 80.

2 Another rule space $\Re_2$ is setup to scan for the *delimiter* in the packet which precedes the host name field. The delimiter field can occur in many ways, and rule space $\Re_2 = \rho_1, \rho_2$ … is set up with multiple rules. For example, ρ_1 = "***[Hh][Oo][Ss][Tt][:][\t\r]**." This searches for the string "host:" insensitive of letter case, followed by any white space.

3 A third rule space $\Re_3$ which contains rules that match the host names is set up. An example of rules in this rule space is = "**www.cnn.com**" and = "**www.yahoo.com**."

4 Similar to rule space $\Re_2$ a rule space R4 is setup to search for the delimiter to the "URL field." Examples of rules which search for this delimiter are $\rho 1$= "***GET:**" and ρ_2 = "***[Pp][Oo][Ss][Tt][:][\t\r]**" and so on.

5 Finally, there are multiple rule spaces $\Re_5$, $\Re_6$, $\Re_7$ … $\Re_n$ which contain rules such as ρ_1 = "**/documents/**," ρ_2 = "**/documents/sports/**," and ρ_3 = "**/music/stream**." A longest prefix match is done to identify the rule that matched.

There are two selection operators, σ_a, the identity function which returns all the rules that matched, used for the first four classifications, and σ_b which defines a longest prefix match operation, used for the final longest prefix on the URL field.

Layer	Algorithm	Time	Application examples
2	Linear search Range search	$O(n)$ $O(\log n)$	Switching
3	Linear search Patricia trie Interval search	$O(n)$ $O(w)$ $O(\log n)$	Forwarding
4	Linear search Tree/Trie search	$O(n)$ $O(\log n)^{d-1}$	Filtering, NAT, QoS Flow ID
7 - Scan, known delimiter	Linear scan/linear search Linear scan/tree search Linear scan/trie search	$O(m+n)$ $O(m+\log n)$ $O(m+w)$	Load balancing, Web caching URL swithcing
7 - Scan, without delimiter	Linear scan/linear search Linear scan/tree search Linear scan/trie search	$O(mn)$ $O(m\log n)$ $O(mw)$	Virus filtering, Intrusion Detection
n is number of rules d is number of dimensions of search m is size of the data part of the packet w is maximum width of the strings			

■ Table 5. *Examples of applications and algorithms.*

There are two different keys set up for the rule spaces. The first key is F_a = {"Destination Port Number"}. The second key is F_b, which points to the whole packet. The start offset of this field is set up dynamically, and the endpoint of the key points to the end of the packet.

Classification Steps — Formally, the pseudocode can be written as follows, and the metrics can be calculated from the definitions in the previous section:

```
/* Definition */
$\Pi_1 =$ if (match ($\pi(\Re_4, F_b, \sigma_a)$) match $\pi(\Re_5, F_b, \sigma_b)$,
$\Pi_2 =$ if (match ($\pi(\Re_4, F_b, \sigma_a)$) match $\pi(\Re_6, F_b, \sigma_b)$,
…,
$\Pi_n =$ if (match ($\pi(\Re_4, F_b, \sigma_a)$) match $\pi(\Re_n, F_b, \sigma_b)$,
/* Execution */
/* Search DIP */
if (match($\pi(\Re_1, F_a, \sigma_a)$) then {
/* Search host delimiter */
  if (match($\pi(\Re_2, F_b, \sigma_a)$) then {
/* Search on host that matched */
  switch($\pi(\Re_3, F_b, \sigma_a)$):($\Pi_1, \Pi_2, \Pi_3 \ldots \Pi_n$)
  }
}
```

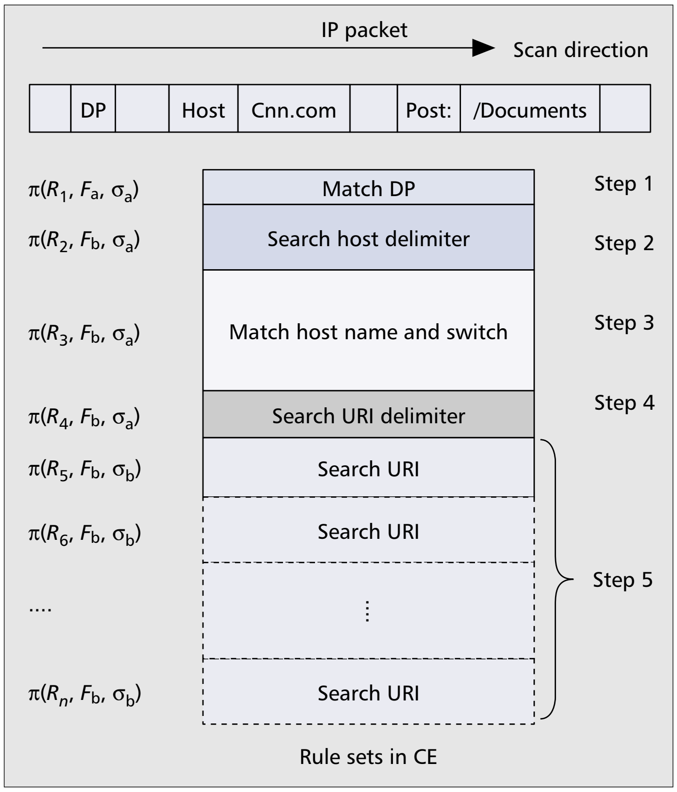

IP packet
Scan direction
DP
Host
Cnn.com
Post:
/Documents
$\pi(R_1, F_a, \sigma_a)$
Match DP
Step 1
$\pi(R_2, F_b, \sigma_a)$
Search host delimiter
Step 2
$\pi(R_3, F_b, \sigma_a)$
Match host name and switch
Step 3
$\pi(R_4, F_b, \sigma_a)$
Search URI delimiter
Step 4
$\pi(R_5, F_b, \sigma_b)$
Search URI
$\pi(R_6, F_b, \sigma_b)$
Search URI
Step 5
....
$\pi(R_n, F_b, \sigma_b)$
Search URI
Rule sets in CE

Figure 4. *The rule sets for server load balancing.*

Other Examples

Table 5 presents some applications along with information regarding the layer at which they perform classification, the algorithms which can be used to implement them on ClassiPI, and the corresponding time complexity metric. In the table, the L7 applications require a linear scan on the packet data of width m for a small number of delimiters (similar to that defined earlier), followed by a search on the rule database. The L7 applications shown in Table 5 are listed separately for both the cases when the delimiter to be searched is known and when there exists no delimiter.

ClassiPI Performance

The performance characteristics of the ClassiPI co-processor for a set of typical classification problems are shown in Fig. 5. These figures are obtained by a simulation on a cycle accurate RTL model of ClassiPI.

The graphs in Fig. 5 show the classification performance of ClassiPI (in million packets per second) for different applications. Since packet arrival rates vary with packet size, the number of packets that arrive per second at Gigabit Ethernet (GE) and OC48c line rates have been overlaid on the graphs. This is convenient for identifying whether the lookup operation performance meets wire-speed requirements. The worst-case performance of the current implementation of ClassiPI is summarized below:

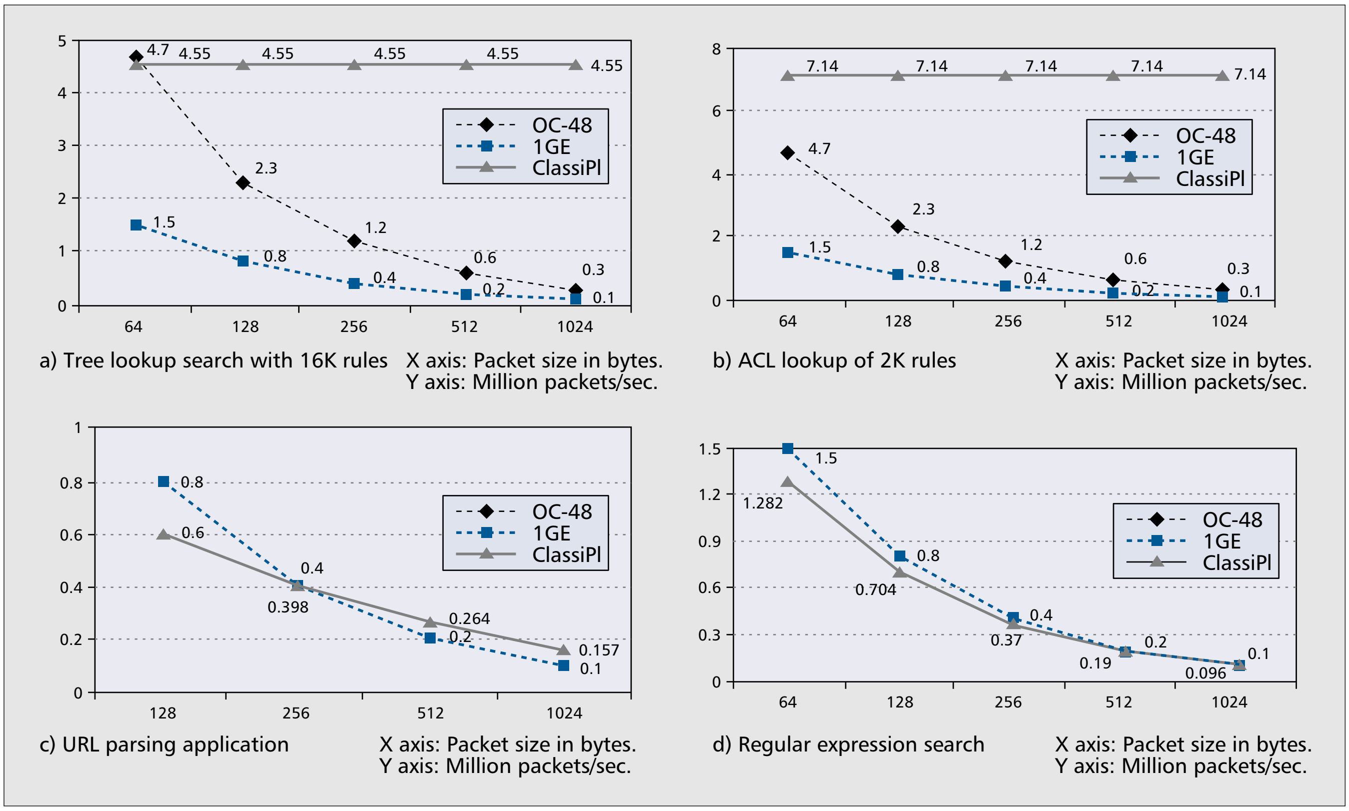
- In Fig. 5a, the performance of ClassiPI for a 2-depth B-tree lookup mechanism is shown. Such a mechanism can be used to implement L2, L3, and L4 searches for a maximum of 16K rules in a single chip. Since only the header fields of the packet need be sent to the device, the performance of the ClassiPI remains constant across all packet sizes. It can be seen that ClassiPI can sustain OC48c requirements across most packet sizes. The two curves converge only for the smallest packet sizes.
- Figure 5b shows the performance of the architecture for ACL-based packet filtering. A maximum of 2K ACL rules per packet are searched in a linear fashion. In this case too, only the packet header needs to be presented to ClassiPI. Hence, packet processing performance is constant across all packet sizes. Again, ClassiPI meets OC48c requirements.
- Figure 5c shows the classification performance for a server load balancing application. A maximum of 256 delimiters, a maximum of 256 distinct server names, and a database of about 750 URLs, as described in an earlier section is assumed.[7] The minimum packet size is specified to be 128 bytes, since packets of smaller sizes are usually combined at the server and aggregated. Note that the application requires the entire packet payload to be scanned, and hence the performance is a function of the packet size. For this operation, ClassiPI maintains a performance curve which closely follows the packet arrival curve for Gigabit Ethernet.
- In Fig. 5d, the performance of ClassiPI is compared for an intrusion detection application. This involves searching for regular expression patterns within the packet payload. For this example, it was assumed that a total of 125 regular expressions had to be parsed for. The performance is again a function of the packet size, and it can be seen that the ClassiPI performance closely matches the Gigabit Ethernet curve.

From the figures it can be seen that the architecture is capable of sustained OC48c line rate performance for L2, L3, and L4 applications and is capable of sustaining 1*GE* throughputs for L7 applications.

Conclusions

The ClassiPI architecture attempts to take advantage of the high speed of CAMs, and the flexibility and scalability of soft-

[7] *These numbers are typical on server load balancing network boxes.*



■ Figure 5. *ClassiPI performance figures.*

ware algorithms. It presents a programmable platform on which appropriate algorithms can be chosen to suit the requirements of specific applications. The current ClassiPI is the first-generation implementation of this architecture, capable of processing packets up to OC48c line rates.

In order to address the needs of emerging network applications, packet classification devices that are flexible and scalable will be required and will continue to evolve. ClassiPI is intended to be one such solution for future network equipment.

Acknowledgments

We would like to thank all the members of the ClassiPI team who helped define its architecture and develop the product. Special thanks are due to Remby Taas who provided the performance figures and helped in reviewing this article.

References

[1] Netlogic Microsystems: http://www.netlogicmicro.com
[2] Lara Networks: http://www.laratech.com
[3] A. Brodnik *et al.*, "Small Forwarding Tables for Fast Routing Lookups," *Proc. ACM SIGCOMM 1997*, Cannes, France, pp. 3–14.
[4] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *Proc. INFOCOM*, San Francisco, CA, vol. 3, Mar.-Apr. 1998, pp. 1248–56.
[5] M. Waldvogel *et al.*, "Scalable High-speed IP Routing Lookups," *Proc. ACM SIGCOMM 1997*, Cannes, France, pp. 25–36.
[6] V. Srinivasan and G. Varghese, "Fast IP Lookups Using Controlled Prefix Expansion," *Proc. ACM Sigmetrics*, June 1998.
[7] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. INFOCOM*, San Francisco, CA, vol. 3, Mar.-Apr. 1998, pp. 1241–48.
[8] T. V. Lakshman and D. Stiliadis, "High Speed Policy Based Packet Forwarding Using Efficient Multi-dimensional Range Matching," *Proc. ACM SIGCOMM*, Vancouver, Canada, Sept. 1998.
[9] M. M. Buddhikot, S. Suri, and M. Waldvogel, "Space Decomposition for Layer 4 Switching," *IFIP Wkshp. Protocols for High Speed Networks*, Salem, MA, Aug. 1999.
[10] A. Feldman and Muthukrishnan, "Tradeoffs for Packet Classification," *Proc. INFOCOM 2000*, Mar. 2000, pp. 1193–1202.
[11] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel, "Fast and Scalable Layer 4 Switching," *Proc. ACM SIGCOMM*, Sept. 1998, pp. 203–14.
[12] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. SIGCOMM, Comp. Commun. Rev.*, vol. 29, no. 4, Sept. 1999, pp. 147-60.
[13] V. Srinivasan *et al.*, "Packet Classification Using Tuple Space Search," *Proc. ACM SIGCOMM*, Sept. 1999.
[14] P. Gupta and N. McKeown, "Classifying Packets Using Hierarchical Intelligent Cuttings," *IEEE Micro*, vol. 20, no. 1, Jan.-Feb. 2000, pp. 34–41.
[15] T. Woo, "A Modular Approach to Packet Classification: Algorithms and Results," *Proc. INFOCOM*, Israel, Mar. 2000.
[16] D. Breslauer, "Efficient String Algorithmics," Ph.D. dissertation, Columbia Univ., CUCS-024-92.
[17] Devavrat Shah and Pankaj Gupta, "Fast Updates on Ternary-CAMs for Packet Lookups and Classification," *Proc. Hot Interconnects VIII*, Stanford Univ., Aug. 2000.

Biographies

SUNDAR IYER (sundar_iyer@pmc-sierra.com) is a senior systems architect at PMC Sierra Inc. He works on the design of hardware architectures for network co-processors. He is also involved in the development of classification algorithms and their implementation. He received a B.Tech. degree in computer science from the Indian Institute of Technology (IIT), Bombay in 1998 and a Master's degree in computer science from Stanford University in 2000. He is presently a doctoral candidate in computer science, in the High Performance Networking Group at Stanford University. His research interests include packet switching algorithms and architectures as well as packet classification and packet buffer design.

RAMANA RAO KOMPELLA (ramana_kompella@pmc-sierra.com) is a software engineer at PMC Sierra Inc. He received a B.Tech. degree in computer science from IIT, Bombay in 1999. He is working toward his Master's degree in computer science at Stanford University. He is an active researcher in the High Performance Networking Group at Stanford University. His research interests include computer networks, computer systems, and high-performance network architectures.

AJIT SHELAT (ajit_shelat@pmc-sierra.com) has over 21 years of engineering development and management experience. Prior to being CTO and co-founder of SwitchOn Networks, he was the founder and managing director of Rimo Technologies Pvt. Ltd., Pune, India; a technology development company that was merged into SwitchOn Networks (now part of PMC-Sierra). Before Rimo, he held a number of senior engineering management positions at Godrej and Boyce, with the company's Electronics Business Equipment division. There he successfully designed and delivered numerous networking and data processing products ranging from access routers to high-end graphics workstations. He graduated with a Bachelor's degree in electrical engineering and a Master's degree in computer science from IIT, Bombay.