Vol. 42 No. 2 IPSJ Journal Feb. 2001

Regular Paper

A Systolic Sieve Array for Real-time Packet Classification

Naohisa Takahashi†

Packet classification, in which the packet header is analyzed and data corresponding to the header is selected, is a key function in implementing routing-table lookups, firewalls, label switching, and differentiated services. This paper presents a packet classifier that can classify packets by using $2 \times n$ memory references in the worst case, assuming the length of the header is n bytes, through a simple operation regardless of the number of classification rules (i.e., filters). Packet classification is modeled as a point-location problem in computational geometry by introducing a function that sieves the filters to reduce an n-dimensional problem to an $n-1$ dimensional problem. A partial evaluator for the sieve function and local and global optimizations are described. A one-dimensional solution to the point location problem can be naturally expanded into a multidimensional problem, and a real-time packet classifier can be implemented for a high-speed network with a relatively small amount of required memory. Preliminary evaluation showed that this classifier can classify packets using only 708 KB of memory and an average of 16.8 memory accesses per packet for 14-byte headers and 10,000 rules.

1. Introduction

As the number of Internet users continues to expand rapidly, networks continue to grow in scale and considerable R&D effort is being focused on high-speed IP routers[1)~3)]. Higher speed is particularly needed to enable routers to perform their primary function — routing-table lookups — faster. Backbone routers that can accommodate 622 Mbps and 2.4 Gbps packets, for example, must be able to quickly search large-scale routing tables (40,000 or more entries).

At the same time there is a cry for greater security, and packet filtering is becoming increasingly important as Internet usage becomes more diversified[4)]. Packet filtering is generally done to ensure security by having the router scrutinize the packet header, and then, for example, relay the packets destined for a web server or discard packets originating from a particular subnetwork, based on the contents of the header. This filtering capability has other functions besides security. It enables service content or quality to be tailored to customers, resulting in differentiated services that have a variable-price structure or variable content. This customization requires switching services or altering the allocation of network resources according to the contents of the packet header.

Among the filtering functions, the ability to analyze the header and determine the appropriate processing is provided by packet classifiers. Packet classifiers are widely used in many network facilities for routing table lookup, packet filtering, avoiding network-wide contention such as the parking-lot problem[5)], and for several kinds of label switching[6)]. The series of conditions for packet classifiers are set as packet-classification rules. A router used in a large-scale network or for fine-grained classification, must be able to handle 1,000 or more such rules. Furthermore, as uses of packet classification proliferate, there will be a growing demand for a longer header field.

In this paper we propose a high-speed packet-classifier that can be applied even when a long header and numerous rules are used. The number of memory references for packet classification is independent of the number of rules. For a header that is n bytes long, packets can be classified by using $2 \times n$ memory references in the worst case.

In this paper we describe a fast method that uses little memory (e.g., less than one megabyte) that is suitable for practical use. This method includes a function that sieves the rules according to a field in the packet header to decompose the packet-classification problem in the same way as for a point-location problem in computational geometry[7)], uses a partial evaluator[8)] for the sieve function, and uses local and global optimizations.

The rest of the paper is organized as follows. We precisely formulate the packet classifier in Section 2. In Section 3, we explain our ap-

† NTT Network Innovation Laboratories

proach, and in Section 4 we describe the partial evaluation method based on partitioning domains and header fields. Section 5 describes the proposed packet-classifier which interprets the data generated by the partial evaluation and by local and global optimizations. In Section 6, we present some preliminary experimental results. In Section 7, we briefly discuss related work.

2. Packet Classifiers

2.1 Functional Overview

The header field that is analyzed to classify packets is called the key field or simply the key. The rules for classifying packets are called filters. A filter consists of any number of predicates that the keys should fulfill. An ordered set of filters is called a filter-set, and is represented by a filter-identifier set. If all keys of a packet P fulfill their associated predicates for a filter f, then f is said to be a feasible filter of P. If any key of P does not fulfill a predicate of a filter f, then f is an infeasible filter with respect to P. A packet classifier is a function that examines whether a key fulfills its associated predicate for all filters and selects a set of all feasible filters. That is, it identifies filter-set $\mathbf{F}'$ that contains all feasible filters in $\mathbf{F}$ pertaining to P when filter-set $\mathbf{F}$ and packet P are given. In cases where only one filter must be determined by routing table lookup or filtering, one filter can be selected from $\mathbf{F}'$ according to the predetermined order of filters (for example, in the description order, or in the order of filter cost specified by the operator). In cases where the filters in $\mathbf{F}$ are arranged in description order, if that same order is preserved in $\mathbf{F}'$, then the first filter in $\mathbf{F}'$ can be selected. If the filters in $\mathbf{F}$ are arranged by prefix length (that is, net-mask length), then longest-prefix-matching is performed. In the implementation described below, it is assumed that when a filter is selected from $\mathbf{F}'$, the filters in $\mathbf{F}$ are arranged in description order.

2.2 Matching Schemes

The predicates can be specified using any of three matching schemes: exact matching, prefix matching, or range matching. Assuming p and x are a predicate and its associated key value, respectively, these schemes are defined as follows.

- Exact matching
  The matching is successful if x equals p.
- Prefix matching
  Assuming v and pl are a prefix and its length, respectively, the upper pl bits of x should be v. In prefix matching for destination IP addresses, for example, the predicate 123.45.67/24 matches the destination IP address whose upper 24 bits are 123.45.67. When multiple predicates are successful, the predicate that has the longest prefix-length is selected in longest-prefix matching.
- Range matching
  Predicates are represented as a set of pairs of keys. Assuming $p = \{[v_{11}, v_{12}], [v_{21}, v_{22}], ..., [v_{n1}, v_{n2}]\}$, if there exists an i such that $v_{i1} \leq x \leq v_{i2}$, the matching of x and p is successful. For the range matching with the predicate $\{[10, 30], [50, 60]\}$, x matches p if $10 \leq x \leq 30$ or $50 \leq x \leq 60$. The predicates for the range matching are useful to specify the conditions that the source or destination port numbers should satisfy.

3. Packet Classifier Using a Filter-sieve Function

3.1 Point-Location Problem

A packet classifier that deals with n keys is modeled as an n-dimensional point-location problem in computational geometry. The n-dimensional point-location problem is described as follows: given a point in n-dimensional space, and a set of m n-dimensional objects, find the object that the point belongs to. In a packet classifier, a packet corresponds to the point and m filters correspond to the m objects. While most conventional point-location problems must deal with the case of non-overlapping objects and use range matching in each dimension, the packet classifier should deal with the case of overlapping filters (i.e., a key might match the corresponding predicates of multiple filters) and use prefix matching and exact matching as well as range matching.

3.2 Filter-sieve Function

A function that uses key value x_k for filter-set $\mathbf{F}$ to remove infeasible filters and return filters whose predicates corresponding to the key are true is called a sieve (**Fig. 1**), represented here by $sieve(\mathbf{F}, k, x_k)$.

The sieve function reduces an n-dimensional point-location problem to an $n-1$ dimensional problem. The result of the packet classifier can be obtained by applying the functions to a filter-set n times.

3.3 SIERRA: Systolic Sieve Array

Figure 2 shows that if all key values are

given as inputs in a pipelined sieve array, a filter-set in which predicates for all keys are true is derived — in other words, the filter-set $\mathbf{F}'$ — as the output of the final stage of the pipeline. This is represented as

$\mathbf{F}' = \mathbf{F_{n-1}}$, provided that
$\mathbf{F_0} = sieve(\mathbf{F}, 0, x_0)$,
$\mathbf{F_1} = sieve(\mathbf{F_0}, 1, x_1)$, ...,
$\mathbf{F_{n-1}} = sieve(\mathbf{F_{n-2}}, n-1, x_{n-1})$.

A packet classifier can be obtained by constructing a pipeline called a systolic sieve array (SIERRA) with sieve processors where each processor executes a sieve function for each key, as shown in **Fig. 3**. In this figure, a "chunk" is a data-set generated by the partial evaluation of a sieve function (described below), while an "instruction" specifies the address of the chunks and is an instruction word that the sieve processor interprets. Given an instruction and a key, a sieve processor reads a chunk and generates the next instruction that will be sent to the sieve processor in the next stage of the pipeline. The processor in the final stage outputs an instruction that includes a pointer to the result of the packet classification. The key values of the incoming packets are successively supplied to the processors (Fig. 3), and each time an instruction word is supplied with the arrival of a packet at the first processor, the system as a whole performs a systolic action[9] within the time it takes for one sieve function to execute. Sieve processors read two chunk memory words to execute one sieve function. By pipelining their memory accesses, we can implement a processor that can produce an output for each memory access.

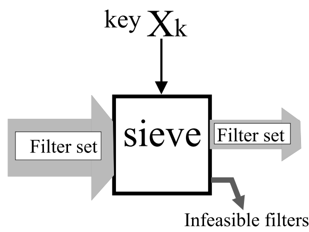

Fig. 1 Sieve: a filter classifier with a key value.

4. Partial Evaluation of a Sieve Function

4.1 Filter-sets

Here, we consider a filter-set, such as that shown below, consisting of filters in which all the fields have been converted to eight-bit keys. This filter-set is represented by $\mathbf{F}$ =

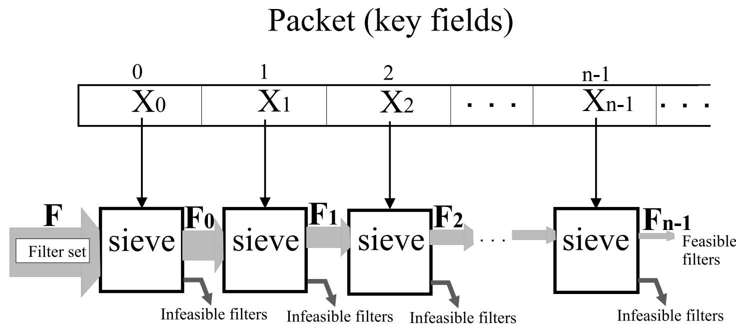

Fig. 2 Finding feasible filters with a sieve array.

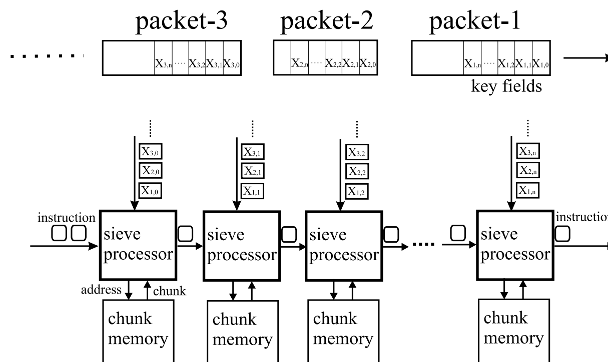

Fig. 3 Conceptual diagram of the systolic sieve array.

$\{1, 2, ..., m\}$. Each element of $\mathbf{F}$ is a filter identifier that is affixed at the beginning of each filter.

$$\begin{array}{rl} 1: & p_{1,1}\ p_{1,2}\ \cdots\ p_{1,n} \\ 2: & p_{2,1}\ p_{2,2}\ \cdots\ p_{2,n} \\ & \vdots \\ f: & p_{f,1}\ p_{f,2}\ \cdots\ p_{f,n} \\ & \vdots \\ m: & p_{m,1} p_{m,2} \cdots\ p_{m,n} \end{array}$$

Here, $p_{f,i}$ is the i-th predicate of f, corresponding to the i-th key. In cases where the packet-field length is not eight bits, the packet-filed is converted to 8-bit key by concatenation or partitioning when necessary as follows.

(1) Flag bit
Multiple bits in the same byte combine to become one key. For example:

$$f_1\colon\ \ p_{1,1}\ [x_1 = 1]\ \ \ p_{1,2}\ [x_2 = 0]$$
$$\Downarrow$$
$$f_{1'}\colon\ \ p_{1',1}\ [x_{1'} = (10)_2]$$

Here, x_1 and x_2 are one-bit flags, and $x_{1'}$ is the value of a one-byte key where bit 0 of $x_{1'}$ is x_2 and bit 1 is x_1.

(2) Multi-byte fields
Multi-byte fields are partitioned into multiple keys. For example:

$$f_1\colon\ \ p_{1,1}\ [x_1 > 300]$$
$$\Downarrow$$
$$f_{1'}\colon\ \ p_{1',1}\ [x_{1'} = 1]\ \ \ p_{1',2}\ [x_{1''} > 44]$$
$$f_{1''}\colon\ \ p_{1'',1}\ [x_{1'} > 1]\ \ \ p_{1'',2}\ [x_{1''} \geq 0]$$

Here, x_1 is the value of a two-byte field and $x_{1'}$ and $x_{1''}$ are the values of one-byte keys where $x_{1'}$ is the upper byte of x_1 and $x_{1''}$ is the lower byte.

(3) Multi-byte prefixes
Multi-byte prefixes are partitioned into multiple keys and are represented by inequalities. For example:

$$f_1\colon\ \ p_{1,1}\ [x_1 = (112*)_{16}]$$
$$\Downarrow$$
$$f_{1'}\colon\ \ p_{1',1}\ [\mathrm{x}_{1'} = (11)_{16}]$$
$$p_{1',2}\ [(20)_{16} \leq \mathrm{x}_{1''} \leq (2\mathrm{f})_{16}]$$

Here, * is a wildcard, x_1 is the value of a two-byte prefix field, and $x_{1'}$ and $x_{1''}$ are the values of one-byte keys where $x_{1'}$ is the upper byte of x_1 and $x_{1''}$ is the lower byte.

Because the key value is $0 \leq x_i \leq 255$, the $p_{f,i}[x_i]$ is represented by a 256-bit vector whose elements are either T ($true$) or F ($false$). Any point on this vector where the value changes from T to F or from F to T is called a boundary; the boundary is represented by the x_i value that follows the change. A pair including the list of all boundaries B and a $p_{f,i}[0]$ value $\langle B, p_{f,i}[0]\rangle$ is called a predicate descriptor; it is used to denote predicate $p_{f,i}$. In the following, this predicate descriptor is used to specify predicates of all rules in the filter-set. This means that filter predicates can be applied to three types of matching schemes: exact matching, prefix matching, or range matching.

In the example shown below, the predicate descriptors of $p_{1,0}$ and $p_{2,0}$ are $\langle(2), F\rangle$ and $\langle(3, 6), F\rangle$, respectively.

[$\mathbf{F_{ex1}} = \{f_1, f_2\}$: Filter-set example 1]

$$\begin{array}{lll} f_1\colon & p_{1,0}\ [x_0 \geq 2] & p_{1,1}\ [x_1 = 6] \\ f_2\colon & p_{2,0}\ [3 \leq x_0 \leq 5] & p_{2,1}\ [x_1 \geq 5] \end{array}$$

4.2 Domain Partitioning

The domain of the key value is partitioned into intervals at all boundaries of all predicates. An ordered set of key values within each interval is called a subdomain. The set of all subdomains has the following properties.

(1) Disjoint: There are no pairs of subdomains that have common elements.
(2) Direct sum: The union of all subdomains equals the original domain.
(3) Unique: When a subdomain is determined, the filter-sets for which a T predicate is given are uniquely determined.

First, consider the partial evaluation of function $sieve(\mathbf{F}, i, x_i)$ by domain partition for the simple filter-set $\mathbf{F_{ex1}}$.

$Sieve(\mathbf{F_{ex1}}, 0, x_0)$ is represented as shown below. When the conditions in parentheses are true, the set in that row is returned as a result.

$$sieve(\{f_1, f_2\}, 0, x_0) = \begin{cases} \{\} & ((\overline{x_0 \geq 2}) \wedge (\overline{3 \leq x_0 \leq 5})) \\ \{f_1\} & ((x_0 \geq 2) \wedge (\overline{3 \leq x_0 \leq 5})) \\ \{f_2\} & ((\overline{x_0 \geq 2}) \wedge (3 \leq x_0 \leq 5)) \\ \{f_1, f_2\} & ((x_0 \geq 2) \wedge (3 \leq x_0 \leq 5)) \end{cases}$$

Partial evaluation (or partial computation) is a systematic method of generating an efficient program based on a given program and some of its data[8]. Partial evaluation of $sieve(\mathbf{F_{ex1}}, 0, x_0)$ essentially means that, before we know the value of x_0, the aforementioned evaluation is pursued as far as possible so that the remaining evaluation needed to derive x_0 is

x_0	0	1	2	3	4	5	6	7
$p_{1,0}$	F	F	T	T	T	T	T	T
	$I^0_{1,0}$		$I^1_{1,0}$					
$p_{2,0}$	F	F	F	T	T	T	F	F
	$I^0_{2,0}$			$I^1_{2,0}$			$I^2_{2,0}$	
Subdomain	D^0_0		D^1_0	D^2_0			D^3_0	
	$\{0,1\}$		$\{2\}$	$\{3,4,5\}$			$\{6,7\}$	
SubDomainFilterSet	$\{\}$		$\{f_1\}$	$\{f_1, f_2\}$			$\{f_1\}$	
DomainDescriptor	0	0	1	2	2	2	3	3

Fig. 4 Example of domain partition.

simplified. The result of the partial evaluation is represented as $sieve_{\mathbf{F_{ex1}},0}(x_0)$.

Consider the following example of a partial evaluation of $sieve(\mathbf{F_{ex1}}, 0, x_0)$. In this example, we assume for simplicity that the key is three bits long. In this case, domain D_0 of x_0 is $D_0 = \{0, ..., 7\}$.

As shown in **Fig. 4**, D_0 can be partitioned into two intervals, $I^0_{1,0}$ and $I^1_{1,0}$, in accordance with the values that can be derived for $p_{1,0}$ (i.e., T or F). Also, D_0 can be partitioned into three intervals based on $p_{2,0}$ in the same way: $I^0_{2,0}$, $I^1_{2,0}$, and $I^2_{2,0}$. When the x_0 domain is partitioned by all interval boundaries, four subdomains are formed: $D^0_0 = \{0, 1\}$, $D^1_0 = \{2\}$, $D^2_0 = \{3, 4, 5\}$, and $D^3_0 = \{6, 7\}$. Scrutinizing the $p_{1,0}$ and $p_{2,0}$ values of each subdomain, a clear filter can be discerned such that when the x_0 value is present in a subdomain, the packet cannot be matched regardless of the value of x_1. For example, F is in both $p_{1,0}$ and $p_{2,0}$, so neither f_1 nor f_2 match regardless of the value of x_1. The subdomain filter-set in Fig. 4 excludes this kind of filter. Moreover, domain descriptors are data that provide subdomain identifiers based on key values, and are derived from a series of subdomain identifiers.

From Fig. 4, $\mathbf{F_0} = sieve_{\{f_1,f_2\},0}(x_0)$ becomes

$$\mathbf{F_0} = \begin{cases} \{\} & (x_0 \in D^0_0 = \{0, 1\}) \\ \{f_1\} & (x_0 \in D^1_0 = \{2\}) \\ \{f_1, f_2\} & (x_0 \in D^2_0 = \{3, 4, 5\}) \\ \{f_1\} & (x_0 \in D^3_0 = \{6, 7\}) \end{cases}$$

4.3 Sieve-Unfolding Tree

Given a filter-set $\mathbf{F}$, subdomains and their filter-sets can be obtained by the partial evaluation of $sieve(\mathbf{F}, 0, x_0)$. Then, for each subdomain, a sieve function that uses the filter-set of the subdomain as an input argument is made. For example, in the case of filter-set $\mathbf{F_{ex1}}$ mentioned above, the function $sieve(\{f_1\}, 1, x_1)$ is made for subdomain $D^1_0 = \{2\}$. Sieve functions for all subdomains are made in this manner and are partially evaluated. As a result, the tree shown in **Fig. 5** can be obtained. It is called a sieve-unfolding tree, and each of its nodes is called a chunk.

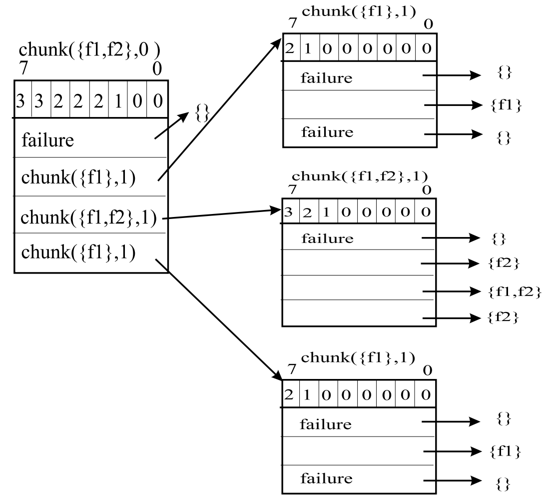

Fig. 5 Sieve unfolding.

A chunk is a data-set that results from a partial evaluation of a sieve function, and it is represented as $chunk(\mathbf{F}, k)$ for $sieve(\mathbf{F}, k, x_k)$. The recursive procedure used to make a sieve-unfolding tree is as follows. Calling this procedure with $\mathbf{F}$ and 0 of the input arguments produces a sieve-unfolding tree in which $chunk(\mathbf{F}, 0)$ is the root.

[$C(\mathbf{F}, k)$: the procedure that makes the $chunk(\mathbf{F}, k)$]

(1) When k is greater than or equal to the number of keys, return to the caller the pointer to $\mathbf{F}$.
(2) Find the subdomain for the k-th predicate of $\mathbf{F}$.
(3) Find filter-set $\mathbf{S}$ that yields true predicates for each subdomain.
(4) Call procedure $C(\mathbf{S}, k + 1)$ for subdomains in which $\mathbf{S}$ is not empty.
(5) Create the data needed to execute sieves (called chunks) by using the results of Steps (2) and (4).

Chunks consist of domain descriptors (DDs)

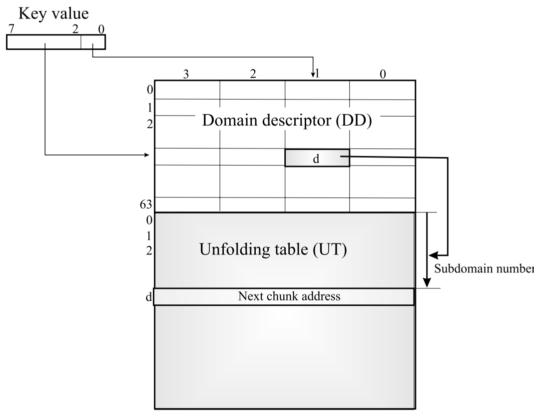

Fig. 6 Naive implementation of a chunk.

and unfolding tables (UTs). The DDs are data used to convert key values into subdomain identifiers, and the UTs provide pointers to sieves for the next key. Because the key values are eight bits long, the values of subdomain identifiers range from 0 to 255. A simple data structure for implementing chunks when using a 32-bit-word memory is shown in **Fig. 6**.

5. Optimization

5.1 Executing Sieve-unfolding Trees

When chunks (Fig. 6) are implemented for all sieves, the sieves are executed when a key is given; one word from the DD corresponding to that key value is read and a word from the UT is read based on that key value; then the sieve of the next key is executed. By repeating these two-word memory references for each sieve (i.e., for each key), one finds whether feasible filters are obtained or not regardless of the number of filters. Note, however, that each chunk requires $256 + 4\delta$ bytes (where δ is the number of subdomains), and that the number of memory references is twice the number of keys. In cases where the number of keys is large (or when the number of generated chunks is large), these problems become serious from a practical standpoint. The next two sections describe local and global optimization methods that overcome these problems.

5.2 Local Optimization

The subdomain identifier can be directly found from the DD when the DD is a vector of subdomain identifiers such as $[0, 0, 1, 2, 2, 2, 3, 3]$, as in Fig. 4. It can also be found from the DD if it includes the bit string $[0, 0, 1, 1, 0, 0, 1, 0]$ representing the interval boundaries (i.e., '1' if the subdomain identifier has been changed and '0' if it has not been changed). In this bit string, for example, one obtains the subdomain identifier associated with each bit by counting the number of 1's from bit 0 to each subsequent bit. This bit string representation makes it possible to compress the domain descriptor by using the following bitmap representation in a manner similar to Degermark's method[10] used to represent prefix trees.

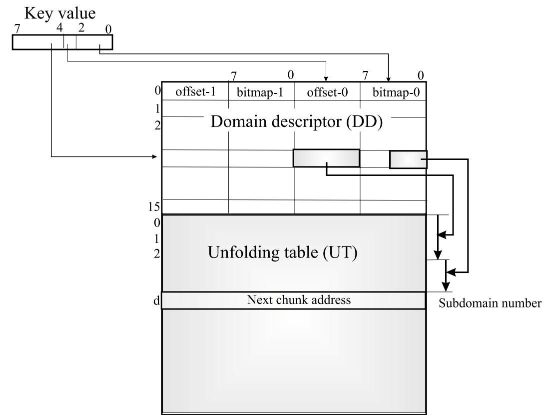

Fig. 7 Optimization 1 (a): bitmap representation for DD.

Figure 7 shows the data structure of a chunk formed from a subdomain identifier obtained by one lookup in the 32-bit-word memory. This domain descriptor uses a pair consisting of a bitmap that can partition the above bit string into eight-bit increments and an offset denoting the subdomain identifier of the initial bit. In this case, one word in the DD is selected by the upper four bits of the key. When bit 3 of the key is a '1', an upper half-word offset and a bitmap are selected; when bit 3 is a '0', a lower half-word offset and a bitmap are selected. Bitmap bit locations are selected using the three least significant bits of the key. The offset is added to the number of 1's from bit 0 to the selected bit in the bitmap, and the resultant value is the subdomain identifier. By using this data structure, we can achieve a DD memory of only 64 bits, just 1/4 the size of the memory needed for the DD shown in Fig. 6.

When the number of subdomains is small, the memory can be reduced even more. For example, **Fig. 8** shows an index-type DD data structure that can be used when the number of subdomains is four or less. In this case, the value of the key providing the subdomain boundary is stored. When the DD is interpreted, the sub-

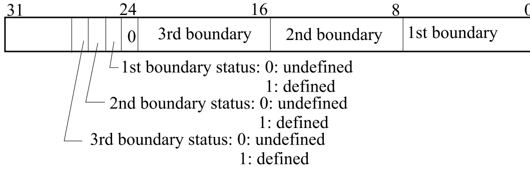

Fig. 8 Optimization 1 (b): index representation for DD.

domain identifier is obtained by comparing the key value against all the boundary values to determine which boundaries the value is between.

5.3 Global Optimization

Redundant chunks in the sieve-unfolding tree are removed to reduce the amount of chunk memory and the number of memory references by analyzing the inter-chunk dependency in the tree.

(1) Sieve share

Figure 4 shows that there are cases where the filter-sets introduced with the next-stage sieve are equivalent even though the subdomains differ. Subdomains of this kind are detected in the partial evaluation of the sieve, and memory is reduced by sharing the next-stage sieve. Moreover, if there are multiple $chunks(\mathbf{F}, k)$ in the sieve-unfolding tree, they are all removed except for one $chunk(\mathbf{F}, k)$ that is shared by their parent chunks.

(2) Sieve skip

Wildcards are commonly used in the representation of filters. When they are, chunks that have only one subdomain might appear in the sieve-unfolding tree. The results of the interpretation of the sieve-unfolding tree do not change even if this type of chunk is skipped, so this optimization reduces the amount of memory and the number of memory references.

(3) Sieve concatenation

In cases where the number of keys is large, only one filter is left after processing the filter-set through a number of stages in the sieve-unfolding tree. In these cases, a determination is made at the following stage to see if the remaining key fulfills all the filter's predicates. Especially in cases where the predicate of the remaining key indicates a specific value and not a range, the values needed to meet the requirements of the key (index) are incorporated into one word by using the DD shown in **Fig. 9** to produce a chunk that can process multiple keys.

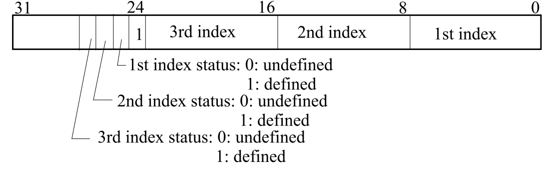

Fig. 9 Optimization 4: DD for sieve concatenation.

This means that multiple chunks are integrated into one chunk, thereby reducing the amount of memory and the number of memory references.

6. Evaluation

6.1 Comparison with Multi-bit Trie

The multi-bit trie[11)] has a simple data structure that enables high-speed packet classification. For example, given the simple filter-set shown below, an 8-bit trie is constructed as shown in **Fig. 10**.

[$\mathbf{F_{ex2}} = \{f_1, f_2, f_3, f_4\}$: Filter-set example 2]

f_1: $p_{1,0}$ $[x_0 = 2]$ $p_{1,1}$ $[x_1 = 1]$
f_2: $p_{2,0}$ $[x_0 = 2]$ $p_{2,1}$ $[x_1 = 2]$
f_3: $p_{3,0}$ $[x_0 = 4]$ $p_{3,1}$ $[x_1 = 3]$
f_4: $p_{4,0}$ $[x_0 = 4]$ $p_{4,1}$ $[x_1 = 4]$

The problem with a multi-bit trie is that it requires a lot of memory in the presence of filters of the form (*, *, port-num); i.e., filters in which the first few fields are wildcards. For example, consider the following filter-set.

[$\mathbf{F_{ex3}} = \{f_1, f_2, f_3, f_4, f_5\}$: Filter-set example 3]

f_1: $p_{1,0}$ $[x_0 = 2]$ $p_{1,1}$ $[x_1 = 1]$ $p_{1,2}$ $[*]$
f_2: $p_{2,0}$ $[x_0 = 2]$ $p_{2,1}$ $[x_1 = 2]$ $p_{2,2}$ $[*]$
f_3: $p_{3,0}$ $[x_0 = 4]$ $p_{3,1}$ $[x_1 = 3]$ $p_{3,2}$ $[*]$
f_4: $p_{4,0}$ $[x_0 = 4]$ $p_{4,1}$ $[x_1 = 4]$ $p_{4,2}$ $[*]$
f_5: $p_{5,0}$ $[*]$ $p_{5,1}$ $[*]$ $p_{5,2}$ $[x_2 = 3]$

The 8-bit trie for $\mathbf{F_{ex3}}$ is shown in **Fig. 11**. On the other hand, a sieve-unfolding tree for $\mathbf{F_{ex3}}$ can be reduced to the small tree with small nodes shown in **Fig. 12** by the local and global optimizations.

6.2 Experimental System

Five types of compilers were implemented. Each compiler was endowed with a different function (as shown below) and generated chunks, which were executable code. In addition, two systems were constructed: an execution system that interpreted the chunks generated by the compilers, and a verification system that checked to make sure the original filter be-

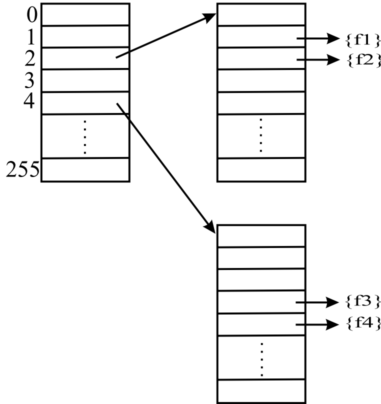

Fig. 10 Multi-bit trie for Example 2.

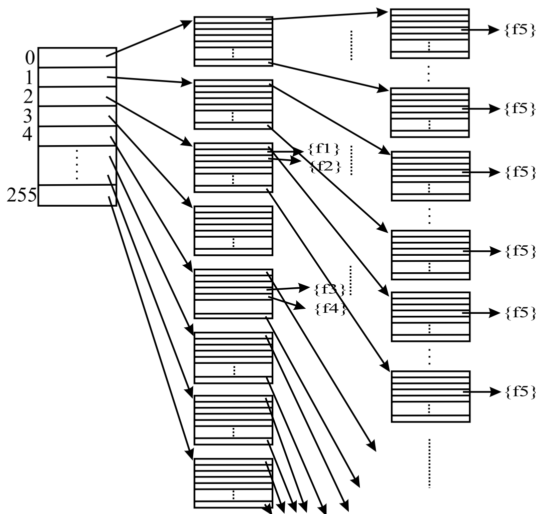

Fig. 11 Multi-bit trie for Example 3.

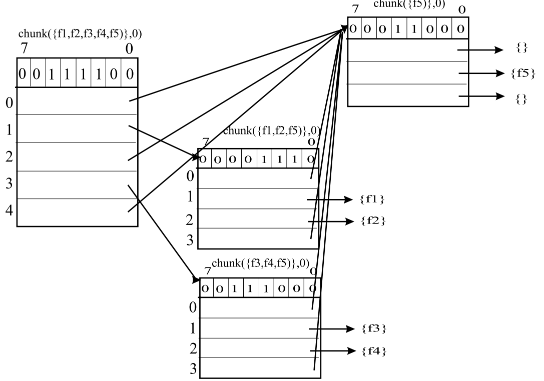

Fig. 12 SIERRA chunks for Example 3.

came a feasible filter when a packet consisting of key values satisfying the predicates of any filter of the original filter-set was generated and executed.

(1) Naive implementation: Without optimization.
(2) Optimization 1: Local optimization.
(3) Optimization 2: (2) + sieve share
(4) Optimization 3: (3) + sieve skip
(5) Optimization 4: (4) + sieve concatenation

6.3 Memory Requirements and References

After generating executable code by using the compilers, the amount of memory and the average number of memory references were determined for three different cases using a single sieve processor and 32-bit-wide memory.

(1) IP routing-table lookup
A filter-set consisting of four-byte keys was configured using a snapshot made on July 20, 1998 of the routing data publicly available on the Web as part of the Internet Performance Measurement and Analysis project[12]. As shown in the first column of **Table 1**, this snapshot included large-scale routing tables, ranging from 3,073 (Paix) to 41,568 (Mae-East) routes. The longest prefix matching was applied to these tables.

(2) Source-destination filtering
First, entries were selected at random from the routing table of Mae-East in Table 1 and regarded as destinations. Multiple source addresses were selected at random from the same routing table and paired with a destination to form a filter. The number of source addresses paired with the same destination was determined randomly.

(3) Packet classification for layer-four switching
In addition to the arrangements given for Case (2), a 14-byte filter was also used; it consisted of a TCP/UDP source and destination port numbers (two bytes each), a protocol field (one byte), and a TCP flag field (one byte)[13]. The probability of a wildcard or a specific value being set in the other fields besides the source and destination fields was 50%. When specific values were used, they were also determined randomly.

The experimental results for each filter-set are presented in Tables 1–3. Table 1 shows that routing-table lookups after Optimizations 3 and 4 used 408 KB of memory with an average of 3.9 memory accesses (8 accesses in the worst

Table 1 Memory requirements and references in IP routing-table lookup.

Filter-set		Naive		Optimization 1		Optimization 2		Optimization 3		Optimization 4	
Name	Length	Size	Ref/P	Size	Ref/P	Size	Ref/P	Size	Ref/P	Size	Ref/P
Mae-East	41,568	15,129	5.7	841	5.7	781	5.7	408	3.9	408	3.9
PacBell	22,837	8,430	5.7	503	5.7	474	5.6	269	3.9	269	3.9
Mae-West	6052	2,473	5.3	134	5.3	131	5.3	63	3.8	63	3.8
Paix	3073	1,232	5.4	92	5.4	91	5.3	63	3.9	63	3.9

Length: number of filters
Size: memory size in KB for all chunks; Ref/P: number of memory references per packet

Table 2 Memory requirements and references in source-destination filtering.

Filter-set		Naive		Optimization 1		Optimization 2		Optimization 3		Optimization 4	
Width (B)	Length	Size	Ref/P	Size	Ref/P	Size	Ref/P	Size	Ref/P	Size	Ref/P
8	100	94	10.9	7	10.9	7	10.9	6	9.8	5	9.5
8	1,000	655	12.3	52	12.3	52	12.3	43	11.1	42	11.1
8	10,000	6,572	12.4	516	12.4	514	12.4	420	11.2	413	11.2

Width: number of key fields; Length: number of filters
Size: memory size in KB for all chunks; Ref/P: number of memory references per packet

Table 3 Memory requirements and references in packet classification for layer-four switching.

Filter-set		Naive		Optimization 1		Optimization 2		Optimization 3		Optimization 4	
Width (B)	Length	Size	Ref/P	Size	Ref/P	Size	Ref/P	Size	Ref/P	Size	Ref/P
14	100	254	17.2	15	17.2	15	17.2	11	16.3	9	15.1
14	1,000	2,240	18.5	124	18.5	124	18.5	91	18.2	71	16.9
14	10,000	22,342	18.5	1,233	18.5	1,231	18.5	900	18.1	708	16.8

Width: number of key fields; Length: number of filters
Size: memory size in KB for all chunks; Ref/P: number of memory references per packet

case), even for the Mae-East filter-set, which contained more than 40,000 entries. When filtering of the source/destination pairs was implemented (**Table 2**), packet classification after optimization 4 was performed even when there were more than 10,000 filters using 413 KB of memory and an average of 11.2 memory accesses (16 accesses in the worst case). When the same filtering conditions were set as in ordinary routing (**Table 3**), the chunk memory was implemented by using 708 KB of memory after optimization 4, even when a 10,000-filter access-control table was used. In this case, the average number of memory accesses was 16.8 (28 in the worst case).

The effects of the optimizations are clearly shown in Tables 1–3. Optimization 1 (local optimization) dramatically reduced the required memory capacity. Optimization 2 (sieve sharing) reduced the memory capacity when the number of filters was large. Optimization 3 (sieve skip) greatly reduced both memory capacity and the number of memory accesses. Finally, Optimization 4 (sieve concatenation) greatly reduced the memory capacity needed for long keys and the number of memory accesses.

Table 4 Execution time for a packet classification using a 450-MHz Pentium II PC.

Filter-set	Width (B)	Length	Time (μs)
Mae-East	4	41,568	0.76
src-dest	8	10,000	1.46
layer-4	14	10,000	2.13

Width: number of key fields
Length: number of filters
Time: execution time per packet

6.4 Execution Time

Although the sieve processor is designed as a dedicated processor to be executed in a pipelined fashion, general-purpose processors like Intel Pentiums can be used to simulate the sieve processor (i.e., interpret the chunks of the sieve-unfolding tree). The execution time for one packet classification using the largest filter-set in each of Tables 1–3 is presented in **Table 4**. Local and global optimizations were applied to the chunks used in the experiment. In this experiment, a PC with a 450-MHz Intel Pentium II processor was used to simulate the sieve processor.

The instruction word for the sieve processor is shown in **Fig. 13**. When simulating the sieve processor, general-purpose processors requires a lot of time for instruction decoding because of

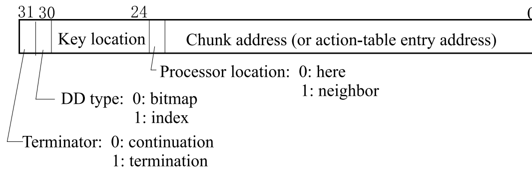

Fig. 13 Instruction word for sieve processors.

the large amount of bit manipulation. However, a real-time (i.e., line-speed) packet classifier can be implemented using a general-purpose processor when the transmission speed of the communication lines is less than 155 Mbps, since a packet can be classified within about 2 μs (i.e., at a rate of about 0.5 Mpps (Mega packets per second)), when the packet length exceeds 40 bytes.

7. Related Work

A number of schemes have been proposed for high-speed searching for longest-matching prefixes[10),14)~17)]. In one approach, for example, associative memory is used for each prefix length, and retrievals are performed as a single concurrent associative access[14)]. This approach is simple, but is difficult to apply when the number of filters or keys is large, since associative memories require much more hardware than random access memories. More recently, a number of powerful, high-speed methods have been proposed in which longest-prefix matching is implemented, not with special hardware, but by software[10),16)]. Degermark, et al.[10)] proposed a sophisticated method that creates a transform of a binary lookup tree into a data structure that can find a route using four table lookups. The representation also turns out to be extremely compact, yielding tables that often fit into the processor L2 cache.

A multi-bit trie, or M-ary trie[11)], requires only n memory references in exact matching of n-byte keys if there are a sufficient number of tables having 256 entries. Extensions of the multi-bit trie have been proposed that can be applied to prefix matching and to multiple key fields[10),15)]. A multi-bit trie with multiple key fields requires a lot of memory, though, when filters of the form (*, *, port-num) are used; i.e., filters in which the first few fields are wildcards. The amount of required memory increases quickly when such wildcards are used since the wildcards mean all entries of the table must have pointers to their child-tables. The packet classifier proposed in this paper can be considered an extended multi-bit trie. It substantially reduces both the amount of memory and the number of memory references used for execution by performing a partial evaluation using key-field and domain partition in advance, and by using local and global optimizations.

The routing-table lookup and filtering problems can be considered a generalized packet-classification problem and handled using a computational geometry approach in the same way as a point-location problem in multidimensional space[18),19)]. A two-dimensional point-location problem is one in which the area where p is located within a predetermined area is found when the coordinates of point p on a map are given[7)]. The coordinates of p are associated with the packet key field, while the map domain is associated with the range defined by the filters. In the case of two dimensions, where m is the number of filters, an algorithm is available for deriving the area in which p belongs; this algorithm uses $0(m)$ memory in $0(\log(m))$ time[7)]. The number of combinations increases rapidly, though, when the number of dimensions is increased. In the case of n dimensions, an algorithm is available for finding a solution using $0(m^n)$ memory in $0(\log(m))$ time[7)]. However, simply applying this approach as is to the packet-classification problem, say a case in which there are 10,000 filters and 4 key fields, would require 10^{12} B (= 1 TB) of memory, so it is not practical.

Lakshman and Stiliadis[18)] proposed a divide-and-conquer approach in which calculations are executed for each dimension in parallel, and results are then combined. This approach uses only one bit to represent a filter in each dimension and executes multiple filters at the same time by using bit-level parallelism. However, $\lceil m/w \rceil$ memory references are needed to process each dimension, where w is the memory word width and m is the number of filters.

The scheme we propose uses a computational geometry approach to treat the packet-classification problem as a point location in a multidimensional space problem. The main difference between it and previous schemes is that the amount of required memory and the computational complexity at execution are substantially reduced by performing a partial evaluation using key-field and domain partition in advance and by using four types of optimizations. As a result, a one-dimensional solution to the point-location problem can be naturally

expanded into a multidimensional problem, and a single, simple computational mechanism can be achieved regardless of the key length.

While the Lakshman scheme[18] does the lookups in parallel and then does an AND operation, our scheme does them in pipelined fashion. In other words, our scheme uses the results of partial evaluations of the initial dimensions to reduce the computation for the next dimension, while the Lakshman scheme treats each dimension independently. Thus, our scheme should require less computation and memory.

Experimental results have shown that only 71 KB of memory is required for 1,000 filters and only 708 KB of memory is needed for 10,000 filters when a 14-byte key consisting of six fields (including two 4-byte IP addresses, two 2-byte port numbers, a 1-byte protocol field, and a 1-byte TCP flag field)[13] is used. These values are small compared to the requirements of the schemes described in Ref. 18) (a 640 KB memory capacity system capable of supporting up to 5 fields and 512 filters) and Ref. 19) (where 3,951 KB of memory is required to support 4 fields and 10,000 filters). Furthermore, the pipelining in a systolic array enables packets to be continuously classified during a single memory reference.

The scheme called "Recursive Flow Classification[20]", or RFC, has been recently presented just after the earlier version of this paper was presented at a workshop[21]. The RFC generalizes the Lakshman sheme and uses both of parallel and piplened fashions. The significant differences between SIERRA and RFC, are: (1) RFC uses all filters for taking the projections of each dimension, while SIERRA uses them only for the first dimension and uses the reduced filter sets that can be obtained by repeating the removal of infeasible filters for the successive dimensions. As a result, the memory requirement of SIERRA can be less than that of RFC, because the more filters are used, the more memory is required. (2) RFC's optimization is based on the merge of the filters, resulting in losing the distinction between each filter. SIERRA preserves the distinction because its optimization is based on the compression and sharing of the data.

Although we should make more analyses of memory requirements and precomputation time, we can state the followings from the results of some experiments we have made using actual routing tables of four major NAPs, i.e., MAE-east, MAE-west, PacBell and Paix. SIERRA can be useful to some applications in which tables do not change frequently and in which the delay of updating the tables is tolerative, though it requires a lot of precomputation time because of its optimization.

8. Conclusion

We have proposed a high-speed packet classification scheme that can be applied even when the headers are long and many filters are used to analyze the header. It uses a one-dimensional array of sieves to analyze each field and to eliminate filters that have no possibility of matching. By processing packets repeatedly through the array, filters with fields that match all keys are detected. The amount of memory and computational complexity at execution are substantially reduced by partially evaluating the sieve function in advance and by using four types of optimization. As a result, a single, simple computational mechanism can be achieved regardless of the key length. Assuming an analysis-field length of n bytes, this mechanism can analyze packets by using $2 \times n$ memory references in the worst case and by a simple operation regardless of the number of filters.

Using less than one megabyte of memory and relatively few memory accesses, this method can perform longest-prefix-entry lookups from routing tables with more than 40,000 entries, and filtering with 14-byte key fields (consisting of source and destination IP addresses, two port numbers, a protocol field, and a TCP-flag field) based on 10,000 filters. Moreover, packets can be continuously classified during a single memory reference by operating the sieve processor as a systolic array.

Execution using the results of sieve-function partial-evaluation is straightforward, so hardware implementation of the processor should be relatively simple. Based on these promising results, we intend to implement the sieve processor on an LSI chip and develop a packet-classification engine that can be incorporated into high-speed switches, routers, workstations, and PCs.

References

1) Asthana, A., Delph, C., Jagadish, H.V. and Krzyzanowski, P.: Towards a Gigabit IP Router, *J. High-Speed Networks*, Vol.1, No.4, pp.281–288 (1992).
2) Tantawy, A., Koufopavlou, O., Zitterbart, M.

and Abler, J.: On the Design of a Multigigabit IP Router, *J. High Speed Networks*, Vol.3, No.3, pp.209–232 (1994).

3) Partridge, C., Carvey, P.P., Burgess, E., et al.: A 50-Gb/s IP Router, *IEEE/ACM Trans. Networking*, Vol.6, No.3, pp.237–248 (1998).

4) Cheswick, W.R. and Bellovin, S.M.: *Firewalls and Internet Security*, Addison-Wesley (1994).

5) Shreedlhar, M. and Varghese, G.: Efficient Fair Queuing Using Deficit Round Robin, *Proc. SIGCOMM 95*, pp.231–242 (1995).

6) Davie, B., Doolan, P. and Rekhter, Y.: Switching in IP Networks: IP Switching, Tag Switching, and Related Technologies, Morgan Kaufmann (1998).

7) de Berg, M., van Kreveld, M., Overmars, M. and Schwarzkoph, O.: *Computational Geometry – Algorithms and Applications*, Springer-Verlag (1997).

8) Futamura, Y., Nogi, K. and Takano, A.: Essence of Generalized Partial Computation, *Theoretical Computer Science*, Vol.90, pp.61–79 (1991).

9) Kung, H.T.: Why Systolic Architectures?, *IEEE Computer*, Vol.15, No.1 (1982).

10) Degermark, M., Brodnik, A., Carlsson, S. and Pink, S.: Small forwarding tables for fast routing lookups, *Proc. SIGCOMM 97*, pp.3–14 (1997).

11) Knuth, D.E.: Sorting and Searching, *The Art of Computer Programing*, Vol.3, Addison-Wesley (1973).

12) Merit Network and Michigan University, Internet Performance Management and Analysis (IPMA) Project.
http://www.merit.edu/ipma/

13) Comer, D.E.: *Internetworking With TCP/IP*, Vol.I, Prentice Hall (1991).

14) McAuley, A.J. and Francis, P.: Fast Routing Table Lookup Using CAMs, *Proc. IEEE INFOCOM '93*, pp.1382–1391 (1993).

15) Doeringer, W., Karjoth, G. and Nassehi, M.: Routing on Longest-Matching Prefixes, *IEEE/ACM Trans. Networking,* Vol.4, No.1, pp.86–97 (1996).

16) Waldvogel, M., Varghese, G., Turner, J. and Plattner, B.: Scalable high-speed IP routing lookups, *Proc. SIGCOMM 97*, pp.25–36 (1997).

17) Srinivasan, V. and Varghese, G.: Faster IP Lookups using Controlled Prefix Expansion, *Proc. ACM Sigmetrics 98*, pp.1–10 (1998).

18) Lakshman, T.V. and Stiliadis, D.: High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching, *Proc. SIGCOMM 98*, pp.203–214 (1998).

19) Srinivasan, V., Varghese, G., Suri, S. and Waldvogel, M.: Fast and Scalable Layer Four Switching, *Proc. SIGCOMM 98,* pp.191–202 (1998).

20) Gupta, P. and McKeown, N.: Packet classification on multiple fields, *Proc. SIGCOMM 99*, pp.147–160 (1999).

21) Takahashi, N.: Real-time packet classification based on the partial evaluation of filter-sieve functions (in Japanese), *Proc. Workshop on Internet Technologies 99*, pp.190–197, JSSST (1999).

(Received May 8, 2000)
(Accepted October 6, 2000)

Naohisa Takahashi was born in 1951. He received B.E. and M.E. degrees in electrical engineering from the University of Electro-Communications, Tokyo, Japan, in 1974 and 1976, respectively. He received a doctorate in computer science in 1987 from Tokyo Institute of Technology. Since 1976 he has been a researcher in the NTT Electrical Communications Laboratories, where he has been engaged in research on parallel processing, software engineering and computer networks. He is currently a leader of Parallel and Distributed Architectures Research Group in NTT Network Innovation Laboratories. Dr. Takahashi is a member of IPSJ, IEICE, JSSST and ACM.