A Packet Classifier Using a Parallel Branching Program Machine

Hiroki Nakahara*, Tsutomu Sasao*, and Munehiro Matsuura*
*Kyushu Institute of Technology, Iizuka, Japan

Abstract—A branching program machine (BM) is a special purpose processor that uses only two kinds of instructions: Branch and output instructions. Thus, the architecture for the BM is much simpler than that for a general purpose processor (MPU). Since the BM uses the dedicated instructions for a special purpose application, it is faster than the MPU. This paper presents a packet classifier using a parallel branching program machine (PBM). To reduce computation time and code size, first, a set of rules for the packet classifier is partitioned into groups. Then, they are evaluated by the PBM in parallel. Also, this paper shows a method to estimate the number of necessary BMs to realize the packet classifier. The PBM32 consisting of 32 BMs has been implemented on an FPGA, and compared with the Intel's Core2Duo@1.2GHz. The PBM32 is 8.1-11.1 times faster than the Core2Duo, and the PBM32 requires only 0.2-10.3 percent of the memory for the Core2Duo.

I. INTRODUCTION

A packet classification [19] is a key technology in the router and the firewall. A packet header includes a protocol number, a source address, a destination address, and a port number. The packet classifier performs a predefined action for a corresponding rule. Applications for the packet classifier include a firewall (FW), an access control list (ACL), and an IP chain for an IP masquerading technique.

Different uses require systems with different performance. Thus, different architecture should be used. In the data centers and the ISPs (Internet Service Providers), the required throughput is more than tens giga bits per second. Thus, CAMs, FPGAs, or ASICs are used. These devices dissipate much power or require a high development cost. On the other hand, in low-end users including SOHO (small office and home office), the embedded processors or the general purpose processors are used. In this research, we consider the packet filter for the low-end users. So, we compare the performance with a general purpose processor or MPU. The throughput for the state-of-the-art packet classifier using the MPU is at most hundreds mega bits per second [4], so it cannot keep up with accelerated speed up of the Internet.

This paper shows a packet classifier using a parallel branching program machine (PBM) [12]. A branching program machine (BM) is a special purpose processor that uses only two instructions [2], [1], [21]. Thus, the BM has simpler architecture than the MPU. Since the BM has the dedicated branch instructions that are frequently used in the packet classifier, it is faster than the MPU. To realize the packet classifier by the PBM, first, a set of rules for the packet classifier is partitioned into groups. Then, they are evaluated by the PBM in parallel.

Table I shows an example of the packet classification table. The rest of the paper is organized as follows: Section 2 defines the packet classifier; Section 3 introduces the PBM; Section 4 shows the realization of the packet classifier using the PBM; Section 5 compares the PBM with the Intel's Core2Duo; and Section 6 concludes the paper.

II. PRELIMINARY

A. Packet Classifier

A packet classification table consists of a set of rules. Each rule has six input fields: Source address (SA), destination address (DA), source port (SP), destination port (DP), protocol number (PRT), and flag number (FLG)1. Also, it generates a rule number (Rule). A field has entries. In this paper, we assume that the rule number is 4 bits. An entry with the largest number has the highest priority. Note that, each field has four bits, rather than the actual number. This field allows a range match: SP and DP are detected by an exact match; PRT and FLG are detected by a range match. A packet classifier detects matched rules using the packet classification table. When two or more rules match, it selects a rule having the highest priority. In this paper, we assume that the rule with the largest number has the highest priority. Note that, any packet matches a default rule whose rule number is zero. Obviously, the default rule has the lowest priority.

Example 2.1: Table I shows an example of the packet classification table, where an asterisk ‘*’ in an entry matches both 0 and 1, while a dash ‘-’ in a field matches any pattern. Note that, each field has four bits, rather than the actual number of bits to simplify the example.

Example 2.2: Consider the packet classification table shown in Table I. The packet header with $SA = 0000, DA = 1010, SP = 8, DP = 8, PRT = TCP$, and $FLG = 1111$

1Practical packet classification tables have a 1 bit flow direction filed. Since we used an open source packet generator ClassBench [20], we ignore it.
matches rule 3, rule 1, and the default rule. Since the rule 3 has the highest priority, the rule 3 is detected.

B. Representation of Entries by Interval Functions

An entry of a rule can be represented by an interval function [16]. First, we define the interval function.

Definition 2.1: [16] Let \( x_j \in \{0, 1\}, X = (x_1, x_2, \ldots, x_n) \), and \( Y = \sum_{i=1}^{n} x_i 2^{i-1} \). An interval function is

\[
IN(X : A, B) = \begin{cases} 
1 & (A \leq Y \leq B) \\
0 & (\text{otherwise}) 
\end{cases}
\]  

where \( A \) and \( B \) are integers that satisfy \( 0 \leq A \leq B \leq 2^n - 1 \).

Next, we represent any entry by the interval function. Suppose that the packet header is represented by 6-tuple \((X_{SA}, X_{DA}, X_{SP}, X_{DP}, X_{PRT}, X_{FLG})\). Since the entry for \( SP \) and \( DP \) is represented by the range match, they can be directly represented by interval functions. When \( A = B \) in Expr. (1), it shows the exact match. Let \( b \) be the protocol number. The entry for \( PRT \) is represented by

\[
IN(X_{PRT} : b, b).
\]  

Similarly, any entry for \( FLG \) can be represented by an interval function. Let \( x_i \in \{0, 1\}, y_i = \ast, v = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m) \), and \( A = \sum_{i=1}^{n} x_i 2^{i-1} \). Any entry for \( SA \) is represented by

\[
IN(X_{SA} : A 2^m, (A + 1) 2^m - 1).
\]  

Similarly, any entry for \( DA \) can be represented by an interval function.

As shown in Example 2.2, multiple rules may match in a packet classification table. To distinguish them, we use a vectorized packet classification function.

Definition 2.2: Let \( k \) be the number of fields, and \( r \) be the number of rules. A vectorized packet classification function \( \vec{F} \) is

\[
\vec{F} = \bigwedge_{i=1}^{r} \bigvee_{j=1}^{k} IN(X_j : A_{(i,j)}, B_{(i,j)}).
\]  

Note that, \( \vec{e}_i \) is an \( r \)-bit unit vector, where only \( i \)-th bit is one, and the other bits are zeros.

Example 2.3: Table II represents entries in Table I using Exprs. (1), (2), and (3). Note that, \( PRT \) is represented by integers: \( TCP = 1, UDP = 2, \) and \( ICMP = 3 \). In Table II, \( \vec{e}_i \) denotes the unit vector corresponding to the rule number.

Example 2.4: By assigning entries shown in Table II to Expr. (4), we have a vectorized packet classification function \( \vec{F} \), where \( k = 6 \) and \( r = 5 \). When a packet header has values \( SA = 0000, DA = 1010, SP = 8, DP = 8, PRT = 1, \) and \( FLG = 1111 \), we have \( \vec{F} = (0, 1, 0, 1) \). It means that rule 3, rule 1, and the default rule are matched.

C. Priority Encoder Function

A packet header may match multiple rules. To detect the rule with the highest priority, we use a priority encoder function. The priority encoder function for \( r \) rules generates a \( \lceil \log_2 r \rceil \)-bit binary number.

Example 2.5: When the vector \( \vec{F} = (0, 1, 0, 1) \) is applied to the priority encoder function shown in Table III, we have \((0, 1, 1)\). This means that the rule 3 is detected.

The priority encoder function can be represented by the interval function.

Example 2.6: Table IV shows an example of the priority encoder function for \( r = 4 \).

By using the vectorized classification function and the priority encoder function, we can realize the packet classifier with the specified priority.

D. Number of Rules

The embedded packet classifier implemented by the general purpose processor [4], [5] uses 100-300 rules [6]. To compare the performance, we also assume that the number of rules is 200.

III. PARALLEL BRANCHING PROGRAM MACHINE[12]

The packet classifier is realized by a parallel branching program machine (PBM). First, each field is converted to a decision diagram. Then, these decision diagrams are evaluated in parallel.

A. MTQDD

An arbitrary \( n \)-variable logic function can be represented by a BDD (Binary Decision Diagram) [3]. An MTBDD (Multi-Terminal Binary Decision Diagram) can evaluate many outputs at a time. Evaluation of the MTBDD requires \( n \) table look-ups. In this paper, we consider that the evaluation time for the BDD is proportional to a longest path length (LPL). Definitions and optimization techniques are shown in [9].

To further speed up the evaluation, an MDD (Multi-Valued Decision Diagram) [8] is used. In the MDD(q), \( q \) variables are grouped to form a \( 2^q \)-valued super variable. Note that a BDD is equivalent to an MDD(1). When the function is represented by an MDD(q), at most \( \lceil \frac{q}{2} \rceil \) table look-ups are necessary to evaluate an input vector [7]. The evaluation time can be

---

**Table III**

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0***</td>
<td>100</td>
</tr>
<tr>
<td>01**</td>
<td>011</td>
</tr>
<tr>
<td>001*</td>
<td>010</td>
</tr>
<tr>
<td>0001</td>
<td>001</td>
</tr>
<tr>
<td>0000</td>
<td>000</td>
</tr>
</tbody>
</table>

**Table IV**

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN(X:SA,15)</td>
<td>100</td>
</tr>
<tr>
<td>IN(X:4,7)</td>
<td>011</td>
</tr>
<tr>
<td>IN(X:2,3)</td>
<td>010</td>
</tr>
<tr>
<td>IN(X:1,1)</td>
<td>001</td>
</tr>
<tr>
<td>IN(X:0,0)</td>
<td>000</td>
</tr>
</tbody>
</table>
TABLE II
PACKET CLASSIFICATION TABLE REPRESENTED BY INTERVAL FUNCTIONS.

<table>
<thead>
<tr>
<th>SA</th>
<th>DA</th>
<th>SP</th>
<th>DP</th>
<th>PRT</th>
<th>FLG</th>
<th>Rule</th>
<th>( \epsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN(( X_{SA},0,8 ))</td>
<td>IN(( X_{DA},12,13 ))</td>
<td>IN(( X_{SP},0,1 ))</td>
<td>IN(( X_{DP},8,9 ))</td>
<td>IN(( X_{PRT},3,5 ))</td>
<td>IN(( X_{FLG},0,15 ))</td>
<td>4</td>
<td>0000</td>
</tr>
<tr>
<td>IN(( X_{SA},0,3 ))</td>
<td>IN(( X_{DA},8,15 ))</td>
<td>IN(( X_{SP},3,8 ))</td>
<td>IN(( X_{DP},6,8 ))</td>
<td>IN(( X_{PRT},1,1 ))</td>
<td>IN(( X_{FLG},15,15 ))</td>
<td>3</td>
<td>0010</td>
</tr>
<tr>
<td>IN(( X_{SA},0,5 ))</td>
<td>IN(( X_{DA},2,2 ))</td>
<td>IN(( X_{SP},11,11 ))</td>
<td>IN(( X_{DP},7,14 ))</td>
<td>IN(( X_{PRT},2,2 ))</td>
<td>IN(( X_{FLG},5,5 ))</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>IN(( X_{SA},0,7 ))</td>
<td>IN(( X_{DA},8,11 ))</td>
<td>IN(( X_{SP},8,9 ))</td>
<td>IN(( X_{DP},4,11 ))</td>
<td>IN(( X_{PRT},1,1 ))</td>
<td>IN(( X_{FLG},0,15 ))</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>IN(( X_{SA},0,15 ))</td>
<td>IN(( X_{DA},0,15 ))</td>
<td>IN(( X_{SP},0,15 ))</td>
<td>IN(( X_{DP},0,15 ))</td>
<td>IN(( X_{PRT},0,15 ))</td>
<td>IN(( X_{FLG},0,15 ))</td>
<td>0</td>
<td>0000</td>
</tr>
</tbody>
</table>

Fig. 1. Example of MTBDD.

Fig. 2. MTQDD derived from MTBDD in Fig. 1.

Q_BRANCH (ADDR0,ADDR1,ADDR2),INDEX,Sел

<table>
<thead>
<tr>
<th>( | = 1 )</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN(( X_{DA},12,13 ))</td>
<td>16</td>
<td>15</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B_BRANCH (ADDR0,ADDR1),INDEX</td>
<td>31</td>
<td>29</td>
<td>28</td>
<td>2221</td>
<td>16</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>DATASET DATA,REG,ADDR</td>
<td>31</td>
<td>29</td>
<td>24</td>
<td>23</td>
<td>16</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>REG</td>
<td>ADDR</td>
<td>DATA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 3. Mnemonics and Internal Representations.

Reduced by increasing \( q \). However, a node for the MDD(\( q \)) requires pointers proportional to \( 2^q \). For many benchmark functions, total memory size for the MDD(2) achieves its minimum [10]. Since MDD(2) has 4 branches, it is denoted by a QDD (Quaternary Decision Diagram). The QDD machine is known to be the best for the area-time complexity [11].

Example 3.8: The program in Fig. 8 evaluates the MTBDD in Fig. 1. Consider the MTQDD shown in Fig. 2. Fig. 5 shows the MTQDD with address assignment for Q_branch instructions, where SEL has the same meaning as Fig. 4. For A6, B_BRANCH instruction is used for an unconditional jump, since the terminal node ‘10’ is already assigned to A3. Thus, the program in Fig. 9 evaluates the MTQDD.

By changing the address and the SEL as shown in Fig. 6, we can remove the unconditional jump. In this way, for the 3-address quaternary branch, we can optimize the code. The number of unconditional jumps can be minimized by an optimization method shown in [17].

C. Branching Program Machine (BM)

Fig. 10 shows a branching program machine (BM). It consists of the instruction memory that stores up to 256 words of 32 bits; the instruction decoder, the program counter (PC), and the register file. In our implementation, two clocks are used to execute each instruction of the BM. Double-rank flip-flops [13] are used to implement the output register. Fig. 7 shows the double-rank flip-flop, where \( L_1 \) and \( L_2 \) are D-latches. The DATASET instruction sends the values...
A0: B_BRANCH (A1,A7), x0
A1: B_BRANCH (A2,A3), x1
A2: DATASET 01,0,A0
A3: B_BRANCH (A4,A5), x2
A4: DATASET 10,0,A0
A5: B_BRANCH (A4,A6), x3
A6: DATASET 00,0,A0
A7: B_BRANCH (A3,A8), x1
A8: B_BRANCH (A6,A5), x2

Fig. 8. Program Code for the MTBDD in Fig. 1.

A0: Q_BRANCH (A2,A2,A5), X0,00
A1: DATASET 01,0,A0
A2: Q_BRANCH (A3,A3,A4), X1,00
A3: DATASET 10,0,A0
A4: DATASET 00,0,A0
A5: Q_BRANCH (A4,A4,A4), X1,10
A6: B_BRANCH (A3,A3), --

Fig. 9. Program Code for the MTQDD in Fig. 5.

into L1 latches by using C Clock. When all the outputs and state variables are evaluated, the values of L1 are sent to L2 latches by using S_Clock.

D. 8_BM

Fig. 11 shows the architecture of the 8_BM consisting of eight BMs. The output registers of BMs are connected to the inputs of the following BMs through programmable routing boxes. Also, each BM can operate independently.

A programmable routing box implements the bitwise AND and the bitwise OR operation. It also implements constant values: In the programmable routing boxes (highlighted with gray in Fig. 11), constant 1s are generated to perform the bitwise AND operation, while constant 0s are generated to perform the bitwise OR operation. Since BMs are connected each other by sharing a register, each BM can send the signal to other BM by one clock within an 8_BM. Since the BM uses two clocks to perform an instruction, the communication delay can be neglected.

E. Parallel Branching Program Machine

Fig. 12 shows the architecture of the parallel branching program machine (PBM) for the packet classifier. The programmable interconnection connects four 8_BMs. The external inputs (packet headers) are sent to the 8_BMs from the network interface (PHY/MAC). Each 8_BM has external outputs connecting to the programmable interconnection and the system BUS. In addition, the host MPU is used to control the whole system.

IV. REALIZATION OF PACKET CLASSIFIER USING PBM

A. Packet Classification Table Implemented by 8_BM

Since the packet classification table has many inputs and outputs, a direct realization by a single MTQDD is infeasible. Our strategy is as follows: First, we partition the set of rules into several groups (Fig. 13, Step 1). Second, we partition each group into six fields (Fig. 13, Step 2). Third, we convert them to the MTQDDs, and load the data to the 8_BM in the PBM (Fig. 13, Step 3). Finally, we use the PBM to evaluate them in parallel.

Theorem 4.1: Consider a vectorized packet classification function $\vec{F}$. Let $k$ be the number of fields, and $r$ be the number of rules, then we have the relation:

$\vec{F} = \bigvee_{i=1}^{r} \bigwedge_{j=1}^{k} \text{IN}(X_j : A_{(i,j)}, B_{(i,j)})$

(Proof) Let $f_{i,j} = \text{IN}(X_j : A_{(i,j)}, B_{(i,j)})$. Then, vectorized packet classification function $\vec{F}$ can be represented by the
By converting the above POS, we have the sum-of-products (POS):

\[ F^r = \bigwedge_{j=1}^{R} \bigvee_{i=1}^{r} \vec{e}_i \cdot \vec{X}_{j: A(i,j), B(i,j)} \]

\[ = (\vec{e}_1 f_{1,1} \lor \vec{e}_2 f_{2,1} \lor \cdots \lor \vec{e}_r f_{r,1}) \]
\[ \land (\vec{e}_1 f_{1,2} \lor \vec{e}_2 f_{2,2} \lor \cdots \lor \vec{e}_r f_{r,2}) \]
\[ \land \cdots \land (\vec{e}_1 f_{1,k} \lor \vec{e}_2 f_{2,k} \lor \cdots \lor \vec{e}_r f_{r,k}) \quad (5) \]

By converting the above POS, we have the sum-of-products (SOP) whose product consists of forms \( f_{a_i, b_j} \), where \( a_i \in \{1, 2, \ldots, r\} \), and \( b_j \in \{1, 2, \ldots, k\} \). As shown in Definition 2.2, \( \vec{e}_i \) is the unit vector whose \( i \)-th element is only one and the other elements are zero. Thus, only the products having the form \( \vec{e}_a f_{a_1,1} \land \vec{e}_a f_{a_2,2} \land \cdots \land \vec{e}_a f_{a,k, k} \) remain, where \( a \in \{1, 2, \ldots, r\} \). Therefore, Expr. (5) can be represented by

\[ F^r = \bigwedge_{j=1}^{R} \bigvee_{i=1}^{r} \vec{e}_i \cdot \vec{X}_{j: A(i,j), B(i,j)} \]

From the interval functions shown in Table II, by Theorem 4.1, Expr. (4) can be converted to

\[ F^r = \bigwedge_{j=1}^{R} \bigvee_{i=1}^{r} \vec{e}_i \cdot \vec{X}_{SA: A_{SA_i}, B_{SA_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{DA: A_{DA_i}, B_{DA_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{SP: A_{SP_i}, B_{SP_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{DP: A_{DP_i}, B_{DP_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{PRT: A_{PRT_i}, B_{PRT_i}} \]

Note that, in Expr. (6), each sum corresponds to a field in the packet classification table. A function representing a sum is a **vectorized field function**. Note that, Expr. (6) is the product of six terms, while the 8-BM consists of eight BMs. To improve the usability of the 8-BM, we decompose each of SA field and DA field into two. Let \( X_{SAE} \) be the odd bits for SA; \( X_{SAO} \) be the even bits for SA; \( X_{DAE} \) be the even bits for DA; and \( X_{DAO} \) be the odd bits for DA. Expr. (6) is converted to the product of eight vectorized field functions as follows:

\[ F^r = \bigwedge_{j=1}^{R} \bigvee_{i=1}^{r} \vec{e}_i \cdot \vec{X}_{SAE: A_{SAE_i}, B_{SAE_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{SAO: A_{SAO_i}, B_{SAO_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{DAE: A_{DAE_i}, B_{DAE_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{DAO: A_{DAO_i}, B_{DAO_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{SP: A_{SP_i}, B_{SP_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{DP: A_{DP_i}, B_{DP_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{PRT: A_{PRT_i}, B_{PRT_i}} \]
\[ \cdot \bigwedge_{i=1}^{r} \bigvee_{j=1}^{R} \vec{e}_i \cdot \vec{X}_{FLG: A_{FLG_i}, B_{FLG_i}} \]

where \( A_{SAE_i}, B_{SAE_i}, A_{SAO_i}, B_{SAO_i}, A_{DAE_i}, B_{DAE_i}, A_{DAO_i}, \) and \( B_{DAO_i} \) are integers. Note that, Expr. (7) is the product of eight sums. Thus, we can efficiently realize Expr. (7) by the 8-BM and the bitwise-AND gate. The programmable routing box shown in Fig. 11 realizes the bitwise-AND gate.
B. Priority Encoder Function Implemented by BM

As shown in Section II-D, we assume that the number of rules is 200. When the number of rules is more than a few hundreds, the number of inputs for the priority encoder functions is too large, so it is too slow to evaluate it by the BM. To realize the priority encoder function compactly, we assume the following conditions:

1. Any pair of rules in the same group are disjoint.
2. Any pair of rules that belong to different groups may intersect.

Since rules are mutually disjoint in a group, the 8_BM can realize it without the priority encoder. On the other hand, since rules in different groups may intersect, an additional BM for the priority encoder function is attached to the outputs of 8_BMs. Since the number of groups is small, the number of inputs for the BM realizing the priority encoder is also small. Thus, the priority encoder function implemented by the BM is fast enough.

C. Packet Classifier Implemented by PBM

Fig. 14 shows the realization of the packet classifier using the PBMSm, where the rules are partitioned into m groups. An 8_BM in the PBMSm realizes a group. The programmable interconnection connects the m 8_BMs, and a BM realizes the priority encoder (In our implementation, m = 4).

V. ANALYSIS OF VECTORIZED FIELD FUNCTIONS

By analyzing the vectorized field function, we can estimate the number of steps for the BM, and the size of hardware. First, we define the region for a vectored field function.

Definition 5.3: Let \( \hat{H}(X) = \sum_{r=1}^{t} \epsilon^{r}I_{N}(X:A_{r}, B_{r}) \) be a vectorized field function, where \( 0 \leq X \leq 2^n - 1 \), and \( r \) be the number of rules (in other words, the number of interval functions). For each value of \( \hat{H} \), we assign a region, which is an interval or a set of intervals in \([0, 2^n - 1]\).

Example 5.9: Fig. 15 (a) shows the relation of intervals and regions for source address (SA). \( \hat{H} \) takes five different values \{0101, 0011, 0001, 1000, 0000\}, and corresponding regions are \([0, 3]\), \([4, 5]\), \([6, 7]\), \([8, 8]\), and \([9, 15]\), respectively. Fig. 15 (b) shows the relation of intervals and regions for destination address (DA). In this case, the number of regions is eight, since \( \hat{H} \) takes eight values \{0000, 0001, 0101, 0111, 1111, 1011, 0011, 0010\}. Note that, the region for \{0000\} consists of two disjoint intervals \([0, 3]\) and \([15, 15]\).

Example 5.9 shows that, when two interval functions have a common element and also none of the intervals are contained by the other, three new regions are produced. For example, for DA shown in Fig. 15 (b), interval functions IN(XDA:6,8) and IN(XDA:8,9) produce three new regions ([6:7], [8:8], and [9:9]). In contrast, for two interval functions, when one contains the other or does not intersect, only two regions are produced. For example, for SA shown in Fig. 15 (a), interval functions IN(XDA:0,3) and IN(XDA:0,7) produce two regions ([0:3] and [4:7]). From above observations, we have the upper bound of the number of regions for the vectorized field function.

Theorem 5.2: A vectorized field function defined by \( s \) interval functions has at most \( 2s \) regions.

(Proof) We prove it by mathematical induction. When \( s = 1 \), the number of regions is at most two. Assume that the number of regions for \( s \) interval functions is \( t \leq 2s \). When we add an additional interval function, at most two new regions increase. Thus, for \( (s + 1) \) interval functions, the total number of regions is at most \( t + 2 \leq 2s + 2 = 2(s + 1) \).

Example 5.10: The DA shown in Fig. 15 (b) has \( s = 4 \) interval functions. The number of regions is eight.

Theorem 5.3: [15] The vectorized field function for FLG (PRT) with \( s \) intervals has at most \( s + 1 \) regions.

Theorem 5.4: [15] The vectorized field function for the address field has \( s \) intervals has at most \( s + 1 \) regions.

To derive the number of nodes for the MTBDD, first, we introduce the decomposition chart.

Definition 5.4: Consider an integer logic function \( F(X):B^n \rightarrow \{0, 1, \ldots, r\} \), where \( B = \{0, 1\} \) and \( X = (x_0, x_1, \ldots, x_{n-1}) \). Let \( X = (X_B, X_F) \) be a partition of \( X \).
Each column is labeled by bound variables $X_B$, while each row is labeled free variables $X_F$. The corresponding chart entry denotes the function value. The number of different column patterns in the decomposition chart is the column multiplicity. A column that has two or more different entries is a non-constant column, while a column that has the same entries is a constant column.

Example 5.11: Fig. 16 (a) shows the decomposition chart for a vectorized field function of the DA shown in Fig. 15 (b), where $X_B = (x_3, x_2, x_1)$, and $X_F = (x_0)$. Note that, the function value is written in decimal number in Fig. 16 (a), while in Fig. 15 (b), that is written in binary number. Columns for $\{011, 100, 111\}$ are the non-constant columns.

The number of nodes for an index of a quasi-reduced MTBDD corresponds to a column multiplicity for a decomposition chart. Also, the column multiplicity is related to the number of regions for the vectorized field function. A non-constant column in a decomposition chart is represented by a number of regions for the vectorized field function. A non-terminal node in the MTBDD corresponds to a column multiplicity for a decomposition chart. Also, the column multiplicity is related to the number of nodes for the MTBDD that realizes the vectorized field function.

Example 5.11: Fig. 16 (a) shows the decomposition chart for the vectorized field function of the DA shown in Fig. 15 (b), where $X_B = (x_3, x_2, x_1)$, and $X_F = (x_0)$. Note that, the function value is written in decimal number in Fig. 16 (a), while in Fig. 15 (b), that is written in binary number. Columns for $\{011, 100, 111\}$ are the non-constant columns.

The number of nodes for an index of a quasi-reduced MTBDD corresponds to a column multiplicity for a decomposition chart. Also, the column multiplicity is related to the number of regions for the vectorized field function. A non-constant column in a decomposition chart is represented by a number of regions for the vectorized field function. A non-terminal node in the MTBDD corresponds to a column multiplicity for a decomposition chart. Also, the column multiplicity is related to the number of nodes for the MTBDD that realizes the vectorized field function.

Lemma 5.1: [14] The number of different column patterns of the vectorized field function $f$ with $t$ regions is at most $t$.

From the above discussion, we have the upper bound of the number of nodes for the MTBDD that realizes the vectorized field function for $r$ rules.

Theorem 5.5: In an arbitrary index for the MTBDD (MTQDD) representing the vectorized field function for $r$ rules, the number of non-terminal nodes is at most $2r$.

(Proof) We prove the case for MTBDDs. The proof for the case of MTQDDs is similar. Consider a vectorized field function consisting of $r$ interval functions. From Theorem 5.2, the number of regions is at most $2r$. From Lemma 5.1, the number of non-constant column patterns is at most $2r$. Since a non-constant column pattern in the decomposition chart corresponds to a non-terminal node in the QRMTBDD, we have the theorem.

Theorem 5.6: Let $n$ be the number of primary inputs, and $r$ be the number of rules for the packet classification table. Then, the number of nodes for the MTQDD representing the vectorized field function is at most

$$\left\lceil \frac{2p - 1}{3} \right\rceil + \left\lceil \frac{n - p}{2} \right\rceil 2r + (r + 1),$$

where $p$ is an integer satisfying $2^p \leq r$.

(Proof) We partition the nodes of the MTBDD into three parts, and enumerate the number of nodes, separately. We assume that the root node has the index $n$, while the terminal node has the index zero. In the upper part, for the indices from $n$ to $n - p + 1$, consider the complete binary tree. Then, the number of nodes is $2^p$. The node for the MTQDD includes $3$ node or one node of the MTBDD (Fig. 17(a)). Thus, the number of the MTQDD nodes in the upper part is at most

$$\left\lceil \frac{2^p - 1}{3} \right\rceil.$$ (8)

As for the middle part, from Theorem 5.5, for each index, the number of non-terminal nodes is at most $2r$ (Fig. 17(b)). Since a node for the MTQDD corresponds to two indices of the MTBDD, for the middle part, the number of nodes for the MTQDD is at most

$$\left\lceil \frac{n - p}{2} \right\rceil 2r.$$ (9)

In the bottom part, from Fig. 17(c), the number of terminal nodes is at most

$$r + 1.$$ (10)

Therefore, from Exprs. (8), (9), and (10), we have the theorem.

From Theorem 5.6, we can derive the upper bound on the number of nodes for the MTQDD for vectorized field function, and also the number of BMs to represent the given packet classification function.

VI. EXPERIMENTAL RESULTS

A. Implementation of PBM32

We implemented the PBM32 on an Altera’s FPGA. To control the PBM32, we attached the embedded processor Nios II/f. We used Altera’s Cyclone III embedded development
kit utilizing Cyclone III: EP3CLS200F780C7N (198,464 LEs, 891 M9Ks), and used Quartus II (v.9.1) synthesis tool. In our implementation, the PBM32 uses 23,105 LEs and 32 M9Ks. Note that, it does not count the hardware resource for the Nios II/f. The maximum clock frequency was 183.42 MHz.

**B. Comparison with Intel’s Core2Duo**

We compared the execution time and code size for the PBM32 with the Intel’s general-purpose processor Core2Duo. We used an Intel’s Core2Duo U7600 (1.2GHz, L1 instruction 32KB, L1 data 32KB, L2 2MB), and OS: Windows XP SP2. To implement a packet filter, first, we generated a packet filter consisting of 200 rules by a command 'db_generator.exe -bc rulefile 200 -0.5 0.1 packetfilterfile' of ClassBench. We loaded the program code for generated QDDs into the PBM32 in the Core2Duo, the code for the BDD is simpler and faster than that for the QDD. So, the Core2Duo emulates BDDs instead of QDDs. We generated the execution code by gcc compiler with optimization option -O3. To obtain the execution time per a test vector, we generated random packet headers, and obtained the average time excluding the time for the reading and writing packet headers.

Table V compares memory size and execution time, where **Rule** denotes the name of packet classifier; **Time** denotes the execution time for a test vector; and **Mem** denotes the memory size. From Table V, as for the performance, the PBM32 is 8.1-11.1 times faster than the Core2Duo, and as for the memory size, the PBM32 requires 9.7-436.2 times less memory than the Core2Duo.

**VII. CONCLUSION AND COMMENTS**

This paper showed a packet classifier using the PBM32. To reduce computation time and code size, first, a set of rules for packet classifier is partitioned into groups. Then, they are evaluated by the PBM32 in parallel. Also, the paper derives the number of BMs to realize a given packet classifier. Rule denotes the name of packet classifier; Mem denotes the memory size. From Table V, as for the performance, the PBM32 is 8.1-11.1 times faster than the Core2Duo, and as for the memory size, the PBM32 requires 9.7-436.2 times less memory than the Core2Duo.

**VIII. ACKNOWLEDGMENTS**

This research is supported in part by the grant of Regional Innovative Cluster Project of MEXT Global Type (the second stage).

**REFERENCES**


