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
Example of the Packet Classification Table.

Input						Output
SA	DA	SP	DP	PRT	FLG	Rule
1000	110*	[0:1]	[8:9]	ICMP	-	4
00**	1***	[3:8]	[6:8]	TCP	1111	3
010*	0010	[3:11]	[7:14]	UDP	0101	2
0***	10**	[8:9]	[4:11]	TCP	-	1
-	-	[0:15]	[0:15]	-	-	0

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, since we consider a realization of the packet classifier for the Internet protocol version 4 (IPv4), SA and DA have 32 bits, DP, SP, and FLG have 16 bits, and PRT has 8 bits. An entry for SA and DA is specified by an IP address; that for SP and DP is specified by a range of a port number; that for PRT is specified by a protocol number; and that for FLG is specified by a bit vector [18]. Thus, SA and DA are detected by **an LPM match**; SP and DP are detected by **a range match**; and PRT and FLG are detected by **an exact 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 $SA = 0000, DA = 1010, SP = 8, DP = 8, PRT = TCP, and\ FLG = 1111$

[1]Practical 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 $x_i \in \{0,1\}$, $X = (x_1, x_2, \ldots, x_n)$, and $Y = \sum_{i=1}^{n} x_i 2^{i-1}$. An **interval function** is

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

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

$$IN(X_{PRT} : b, b). \quad (2)$$

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

$$IN(X_{SA} : A2^m, (A+1)2^m - 1). \quad (3)$$

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

$$\vec{F} = \bigvee_{i=1}^{r} \vec{e}_i \bigwedge_{j=1}^{k} IN(X_j : A_{(i,j)}, B_{(i,j)}). \quad (4)$$

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. ■.

TABLE III
AN EXAMPLE OF THE PRIORITY ENCODER FUNCTION.

input	output
1***	100
01**	011
001*	010
0001	001
0000	000

TABLE IV
PRIORITY ENCODER FUNCTION REPRESENTED BY INTERVAL FUNCTION WITH NATURAL BINARY ORDERED NUMBER.

input	output
IN(X:8,15)	100
IN(X:4,7)	011
IN(X:2,3)	010
IN(X:1,1)	001
IN(X:0,0)	000

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 $\lceil \frac{n}{q} \rceil$ table look-ups are necessary to evaluate an input vector [7]. The evaluation time can be

TABLE II
PACKET CLASSIFICATION TABLE REPRESENTED BY INTERVAL FUNCTIONS.

SA	DA	SP	DP	PRT	FLG	Rule	$\vec{e}_i$
IN(X_{SA}:8,8)	IN(X_{DA}:12,13)	IN(X_{SP}:0,1)	IN(X_{DP}:8,9)	IN(X_{PRT}:3,3)	IN(X_{FLG}:0,15)	4	1000
IN(X_{SA}:0,3)	IN(X_{DA}:8,15)	IN(X_{SP}:3,8)	IN(X_{DP}:6,8)	IN(X_{PRT}:1,1)	IN(X_{FLG}:15,15)	3	0100
IN(X_{SA}:4,5)	IN(X_{DA}:2,2)	IN(X_{SP}:3,11)	IN(X_{DP}:7,14)	IN(X_{PRT}:2,2)	IN(X_{FLG}:5,5)	2	0010
IN(X_{SA}:0,7)	IN(X_{DA}:8,11)	IN(X_{SP}:8,9)	IN(X_{DP}:4,11)	IN(X_{PRT}:1,1)	IN(X_{FLG}:0,15)	1	0001
IN(X_{SA}:0,15)	IN(X_{DA}:0,15)	IN(X_{SP}:0,15)	IN(X_{DP}:0,15)	IN(X_{PRT}:0,15)	IN(X_{FLG}:0,15)	0	0000

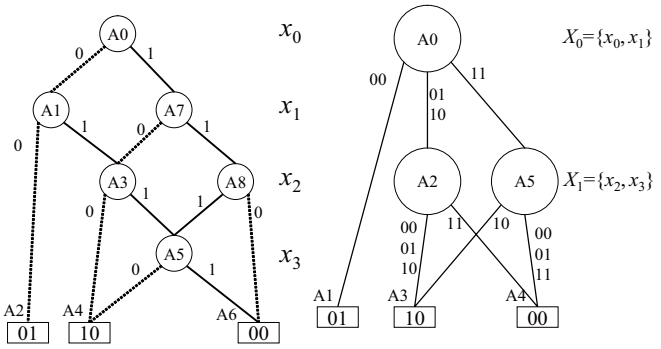

Fig. 1. Example of MTBDD.

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

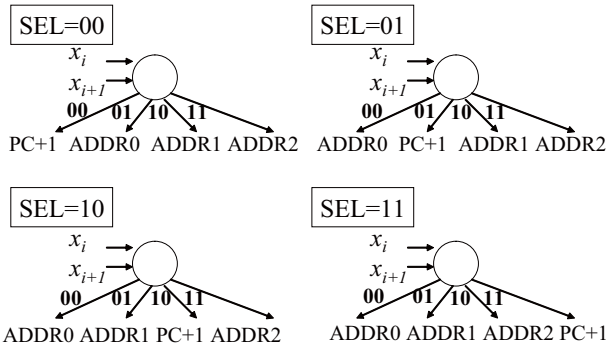

Fig. 4. Four Different Q_BRANCH Instructions

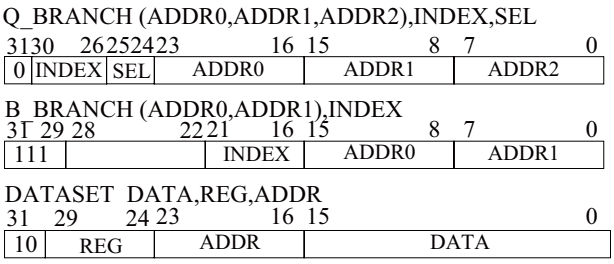

Fig. 3. Mnemonics and Internal Representations.

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 3.7: Fig.1 shows an example of the MTBDD. Fig. 2 shows the MTQDD that is derived from the MTBDD in Fig. 1. ■

B. Instructions for the Branching Program Machine [17]

Three instructions are used to evaluate an MTQDD. A **2-address binary branch instruction** (B_BRANCH) and a **3-address quaternary branch instruction** (Q_BRANCH) evaluate a non-terminal node, while a **dataset instruction** (DATASET) evaluates a terminal node. Mnemonics and their internal representations for *B_BRANCH*, *Q_BRANCH* and *DATASET* are shown in Fig. 3.

B_BRANCH performs a binary branch: If the value of the variable specified by INDEX is equal to 0, then GOTO ADDR0, else GOTO ADDR1. **DATASET** performs an output operation and a jump operation. First, *DATASET* writes DATA (16 bits) to a register specified by REG. Then, GOTO ADDR. **Q_BRANCH** jumps to one of four addresses: Three jump addresses are specified by *ADDR0*, *ADDR1*, and *ADDR2*, while the remaining address is the next address (*PC+1*) to the present one. Since it evaluates two variables at a time, the total evaluation time is reduced up to a half of a *B_BRANCH* instruction. Also, it can reduce the total number of instructions. We use four different *Q_BRANCH* instructions shown in Fig. 4. *SEL* in the *Q_BRANCH* specifies one of four combinations. Let i be the value of the variable specified by *INDEX*. If (*SEL=i*), then jump to *PC+1*, otherwise jump to $ADDR_i$. In addition, **unconditional jump instructions** are necessary to evaluate some QDDs. The next Example illustrates this:

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 quarternary 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 filp-flops** [13] are used to implement the output register. Fig. 7 shows the double-rank filp-flop, where L_1 and L_2 are D-latches. The *DATASET* instruction sends the values

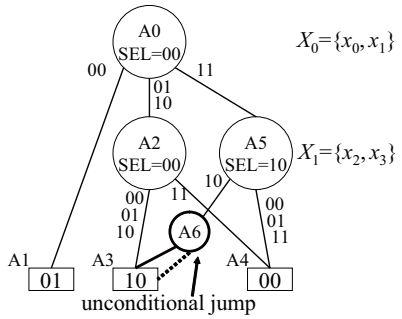

Fig. 5. MTQDD with 3-address Quaternary Branch Instructions.

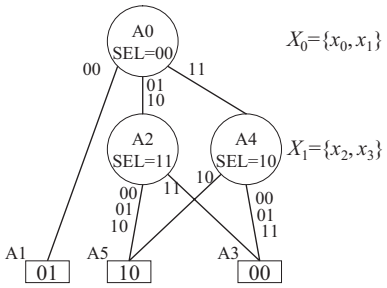

Fig. 6. Optimal Assignment of Fig. 5.

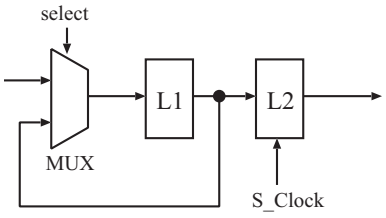

Fig. 7. Double-Rank Flip-Flop.

```
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.

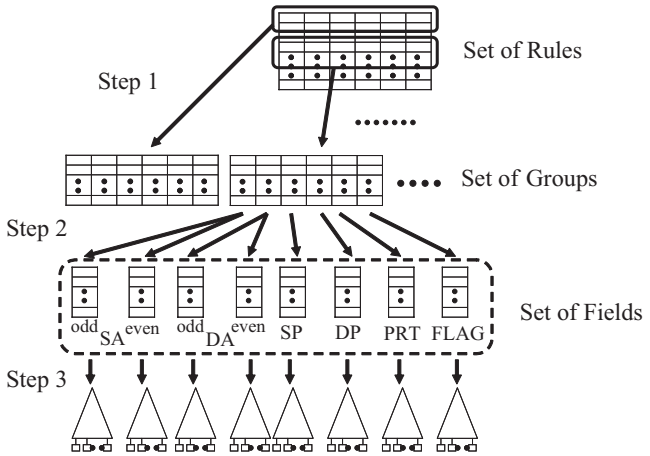

Fig. 13. Partition of Packet Classification Table.

into L_1 latches by using **C_Clock**. When all the outputs and state variables are evaluated, the values of L_1 are sent to L_2 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:

$$\begin{aligned}\vec{F} &= \bigvee_{i=1}^{r} \vec{e}_i \bigwedge_{j=1}^{k} IN(X_j : A_{(i,j)}, B_{(i,j)}) \\ &= \bigwedge_{j=1}^{k} \bigvee_{i=1}^{r} \vec{e}_i IN(X_j : A_{(i,j)}, B_{(i,j)})\end{aligned}$$

(Proof) Let $f_{i,j} = IN(X_j : A_{(i,j)}, B_{(i,j)})$. Then, vectorized packet classification function $\vec{F}$ can be represented by the

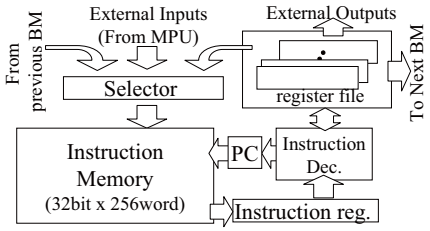

Fig. 10. Branching Program Machine (BM).

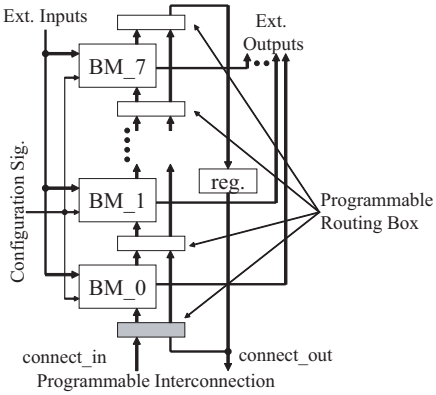

Fig. 11. Architecture of 8_BM.

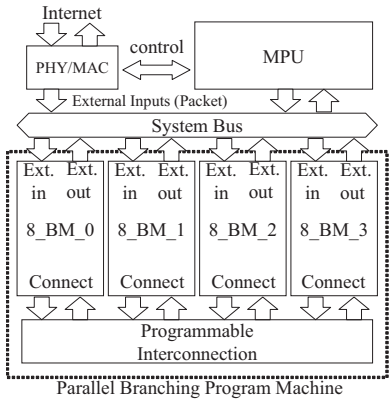

Fig. 12. System of Packet Classifier Using Parallel Branching Program Machine (PBM).

product-of-sums (POS):

$$\begin{aligned}\vec{F} &= \bigwedge_{j=1}^{k}\bigvee_{i=1}^{r}\vec{e}_i IN(X_j : A_{(i,j)}, B_{(i,j)}) \\ &= (\vec{e}_1 f_{1,1} \vee \vec{e}_2 f_{2,1} \vee \cdots \vee \vec{e}_r f_{r,1}) \\ &\quad \wedge (\vec{e}_1 f_{1,2} \vee \vec{e}_2 f_{2,2} \vee \cdots \vee \vec{e}_r f_{r,2}) \\ &\quad \wedge \cdots \wedge (\vec{e}_1 f_{1,k} \vee \vec{e}_2 f_{2,k} \vee \cdots \vee \vec{e}_r f_{r,k}). \end{aligned} \quad (5)$$

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 $\vec{e}_\alpha f_{\alpha,1} \wedge \vec{e}_\alpha f_{\alpha,2} \wedge \cdots \wedge \vec{e}_\alpha f_{\alpha,k}$ remain, where $\alpha \in \{1,2,\ldots,r\}$. Therefore, Expr. (5) can be represented by

$$\begin{aligned}\vec{F} &= \vec{e}_1(f_{1,1} \wedge f_{1,2} \wedge \cdots \wedge f_{1,k}) \\ &\quad \vee \vec{e}_2(f_{2,1} \wedge f_{2,2} \wedge \cdots \wedge f_{2,k}) \\ &\quad \vee \cdots \vee \vec{e}_r(f_{r,1} \wedge f_{r,2} \wedge \cdots \wedge f_{r,k}) \\ &= \bigvee_{i=1}^{r}\vec{e}_i \bigwedge_{j=1}^{k} IN(X_j : A_{(i,j)}, B_{(i,j)}).\end{aligned}$$

□

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

$$\begin{aligned}\vec{F} &= \bigvee_{i=1}^{r}\vec{e}_i IN(X_{SA} : A_{SA_i}, B_{SA_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{DA} : A_{DA_i}, B_{DA_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{SP} : A_{SP_i}, B_{SP_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{DP} : A_{DP_i}, B_{DP_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{PRT} : A_{PRT_i}, B_{PRT_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{FLG} : A_{FLG_i}, B_{FLG_i}). \end{aligned} \quad (6)$$

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 even bits for SA; X_{SAO} be the odd 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:

$$\begin{aligned}\vec{F} &= \bigvee_{i=1}^{r}\vec{e}_i IN(X_{SAE} : A_{SAE_i}, B_{SAE_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{SAO} : A_{SAO_i}, B_{SAO_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{DAE} : A_{DAE_i}, B_{DAE_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{DAO} : A_{DAO_i}, B_{DAO_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{SP} : A_{SP_i}, B_{SP_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{DP} : A_{DP_i}, B_{DP_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{PRT} : A_{PRT_i}, B_{PRT_i}) \\ &\quad \cdot \bigvee_{i=1}^{r}\vec{e}_i IN(X_{FLG} : A_{FLG_i}, B_{FLG_i}), \end{aligned} \quad (7)$$

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.

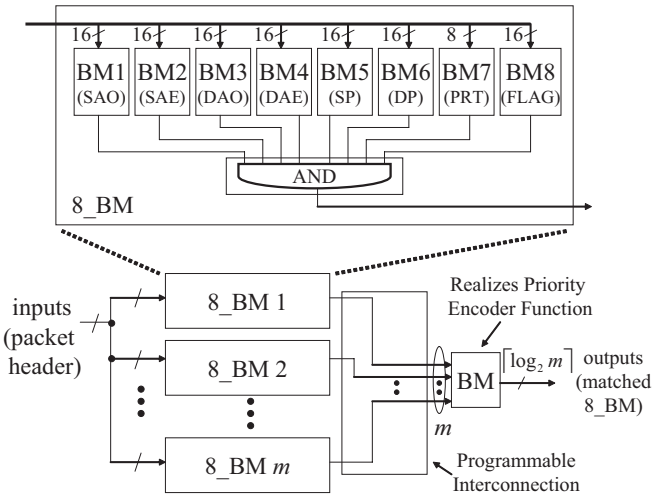

Fig. 14. Realization of the Packet Classifier Using PBM.

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].
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 PBM$8m$, where the rules are partitioned into m groups. An 8_BM in the PBM$8m$ 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 $\vec{H}(X) = \bigvee_{i=1}^{r} \vec{e}_i IN(X : A_i, B_i)$ 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 $\vec{H}$, we assign a **region**, which is an interval or a set of intervals in $[0, 2^n - 1]$.

[2]Our tool converts the set of rules into disjoint ones to satisfy this condition.

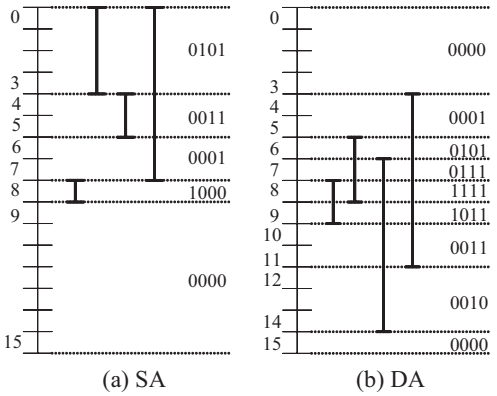

Fig. 15. Relation of Interval and Region.

Example 5.9: Fig. 15 (a) shows the relation of intervals and regions for source address (SA). $\vec{H}$ 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 $\vec{H}$ 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(X_{DA}:6,8) and IN(X_{DA}: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(X_{DA}:0,3) and IN(X_{DA}: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.

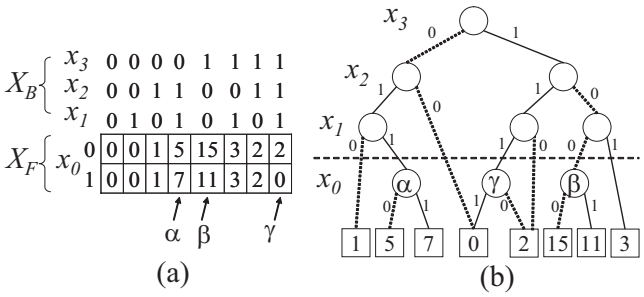

Fig. 16. Relation Between the Decomposition Chart and MTBDD.

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 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 non-terminal node in the MTBDD. For example, in Fig. 16, non-constant columns (α, β, and γ) correspond to nodes (α, β, and γ), respectively. In contrast, the constant columns correspond to the terminal nodes. Thus, the number of the different non-constant columns equals to the number of nodes for the corresponding index of the quasi-reduced MTBDD.

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.

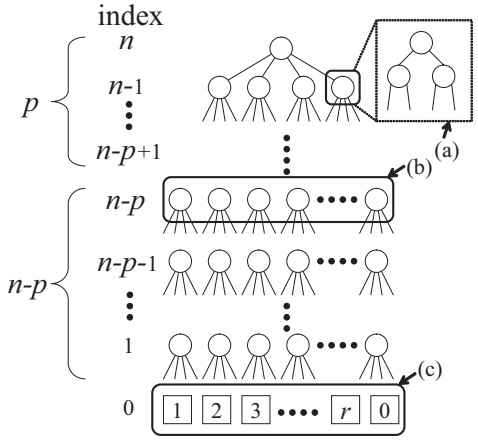

Fig. 17. Explanation of Theorem 5.6.

Then, the number of nodes for the MTQDD representing the vectorized field function is at most

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

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

(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

$$\lceil \frac{2^p - 1}{3} \rceil. \tag{8}$$

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

$$\lceil \frac{n-p}{2} \rceil 2r. \tag{9}$$

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

$$r + 1. \tag{10}$$

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

TABLE V
COMPARISON OF PBM32 WITH INTEL'S CORE2DUO.

Rule	PBM32		Core2Duo		Ratio (C2D/PBM)	
	Time [nsec]	Mem [KB]	Time [nsec]	Mem [KB]	Time	Mem
acl1	98	8.9	945	86.6	9.6	9.7
acl2	98	7.8	945	127.7	9.6	16.3
acl3	98	10.1	801	143.9	8.1	14.2
acl4	98	10.0	801	138.3	8.1	13.7
acl5	98	7.9	945	107.2	9.6	13.5
fw1	98	3.0	1089	624.6	11.1	203.7
fw2	98	4.5	801	261.8	8.1	57.6
fw3	98	1.8	1089	708.6	11.1	379.5
fw4	98	2.6	945	538.5	9.6	202.7
fw5	98	2.5	1089	1104.1	11.1	436.2
ipc1	98	12.9	1089	142.6	11.1	11.0
ipc2	98	1.4	1089	67.5	11.1	46.6

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, Cache L1 data 32KB, L1 instruction 32KB, and L2 2MB), and OS: Windows XP SP2. To implement a packet filter, first, we generated a packet filter consisting of 200 rules by using a command 'db_generator.exe -bc *rulefile* 200 2 -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 that for 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 derived the number of BMs to realize a given packet classifier. We implemented the PBM32 on an FPGA, and compared it with the Intel's Core2Duo@1.2GHz microprocessor. 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.

VIII. ACKNOWLEDGMENTS

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

REFERENCES

[1] P. C. Baracos, R. D. Hudson, L. J. Vroomen, and P. J. A. Zsombor-Murray, "Advances in binary decision based programmable controllers," *IEEE Transactions on Industrial Electronics,* Vol. 35, No. 3, pp. 417-425, Aug., 1988.

[2] R. T. Boute, "The binary-decision machine as programmable controller," *Euromicro Newsletter,* Vol. 1, No. 2, pp. 16-22, 1976.

[3] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, Vol. C-35, No. 8, pp. 677-691, Aug. 1986.

[4] CISCO: ASA5500 series, http://www.cisco.com/

[5] K. Golnabi, R. K. Min, L. Khan, L. and E. Al-Shaer, "Analysis of Firewall Policy Rules Using Data Mining Techniques," *NOMS2006*, pp. 305-315, April, 2006.

[6] P. Gupta and N. McKeown, "Packet classification on multiple fields," *ACM Sigcomm*, August, 1999.

[7] Y. Iguchi, T. Sasao, and M. Matsuura, "Implementation of multiple-output functions using PROMDDs," *ISMVL2000*, Portland, Oregon, U.S.A., Mya 23-25, 2000, pp.199-205.

[8] T. Kam, T. Villa, R. K. Brayton, and A. L. Sagiovanni-Vincentelli, "Multi-valued decision diagrams: Theory and Applications," *Multiple-Valued Logic: An International Journal,* Vol. 4, No. 1-2, pp. 9-62, 1998.

[9] S. Nagayama and T. Sasao, "On the minimization of longest path length for decision diagrams," *IWLS2004*, June 2-4, Temecula, California, U.S.A., pp. 28-35.

[10] S. Nagayama, T. Sasao, Y. Iguchi, and M. Matsuura, "Area-time complexities of multi-valued decision diagrams," *IEICE Transactions on Fundamentals of Electronics,* Vol. E87-A, No. 5, pp. 1020-1028, May 2004.

[11] H. Nakahara, T. Sasao, and M. Matsuura, "A comparison of architectures for various decision diagram machines," (to be published).

[12] H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, "Emulation of sequential circuits by a parallel branching program machine," *ARC2009*, Karlsruhe, Germany, March 16-18, 2009. *LNCS5443*, pp. 261-267, March 2009.

[13] T. Sasao, H. Nakahara, M. Matsuura and Y. Iguchi, "Realization of sequential circuits by look-up table ring," *MWSCAS2004*, Hiroshima, pp.I:517-I:520, July 25-28, 2004.

[14] T. Sasao, J. T. Butler, and M. D. Riedel, "Application of LUT cascades to numerical function generators," *SASIMI2004*, pp.422-429, Oct. 18-19, 2004.

[15] T. Sasao and J. T. Butler, "Implementation of multiple-valued CAM functions by LUT cascades," *ISMVL2006*, May, 2006.

[16] T. Sasao, "On the complexity of classification functions," *ISMVL2008*, May 22-24, 2008.

[17] T. Sasao, H. Nakahara, M. Matsuura, Y. Kawamura, and J.T. Butler, "A quaternary decision diagram machine and the optimization of its code," *ISMVL2009*, May, 2009, pp.362-369.

[18] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended TCAMs," *ICNP2003*, pp. 120-131, Nov., 2003.

[19] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys,* Vol. 37, Issue 3, pp. 238-275, Sep., 2005.

[20] D. E. Taylor and J. S. Turner, "ClassBench: a packet classification benchmark," *INFOCOM2005*, Vol. 3, pp. 2068-2079, 13-17, March, 2005.

[21] P.J.A.Zsombor-Murray, L.J. Vroomen, R.D. Hudson, Le-Ngoc Tho, and P.H. Holck, "Binary-decision-based programmable controllers, Part I-III," *IEEE Micro* Vol. 3, No. 4, pp. 67-83 (Part I), No. 5, pp. 16-26 (Part II), No. 6, pp. 24-39 (Part III), 1983.