IEICE TRANS. INF. & SYST., VOL.E93–D, NO.8 AUGUST 2010

PAPER Special Section on Multiple-Valued Logic and VLSI Computing

# A Parallel Branching Program Machine for Sequential Circuits: Implementation and Evaluation

**Hiroki NAKAHARA**[†a)], **Tsutomu SASAO**[†b)], **Munehiro MATSUURA**[†c)], *and* **Yoshifumi KAWAMURA**[††d)], ***Members***

**SUMMARY** The parallel branching program machine (PBM128) consists of 128 branching program machines (BMs) and a programmable interconnection. To represent logic functions on BMs, we use quaternary decision diagrams. To evaluate functions, we use 3-address quaternary branch instructions. We realized many benchmark functions on the PBM128, and compared its memory size, computation time, and power consumption with the Intel's Core2Duo microprocessor. The PBM128 requires approximately a quarter of the memory for the Core2Duo, and is 21.4-96.1 times faster than the Core2Duo. It dissipates a quarter of the power of the Core2Duo. Also, we realized packet filters such as an access controller and a firewall, and compared their performance with software on the Core2Duo. For these packet filters, the PBM128 requires approximately 17% of the memory for the Core2Duo, and is 21.3-23.7 times faster than the Core2Duo.

***key words:*** *embedded system, branching program machine, multi-processing, BDD*

## 1. Introduction

A branching program machine (BM) is a special-purpose processor that evaluates binary decision diagrams (BDDs) [2], [3], [25]. The BM 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 microprocessor (MPU). Since the BM uses the dedicated instructions to evaluate BDDs, it is faster than the MPU. In fact, for control applications, the BM is much faster than the MPU [2]. The applications of BMs include sequencers [3], [25], logic simulators [1], [11], [19], and packet filters for the Internet [22].

In this paper, we show the parallel branching program machine (PBM128) that consists of 128 BMs and a programmable interconnection. To reduce computation time and code size, we use special instructions that evaluate consecutive two nodes at a time. To evaluate code size and computation time for the PBM128, we compare with the Intel's general-purpose processor Core2Duo. In this paper, we implement packet filters as well as a logic simulator.

Manuscript received November 10, 2009.
Manuscript revised March 17, 2010.

[†]The authors are with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka-shi, 820–8502 Japan.
[††]The author is Renesas Technology Corp., Tokyo, 100–0004 Japan.
a) E-mail: nakahara@aries01.cse.kyutech.ac.jp
b) E-mail: sasao@cse.kyutech.ac.jp
c) E-mail: matsuura@cse.kyutech.ac.jp
d) E-mail: kawamura.yoshifumi@renesas.com

DOI: 10.1587/transinf.E93.D.2048

The rest of the paper is organized as follows: Sect. 2 introduces the BM that emulates a sequential circuit. Section 3 presents an architecture of the PBM128. Section 4 shows the implementation of the PBM128, and compares the PBM128 with the general-purpose MPU. Section 5 shows a realization of packet filters. Finally, Sect. 6 concludes the paper.

This paper builds on the previous publication [18].

## 2. Branching Program Machine to Emulate Sequential Circuits

In this section, we show the branching program machine (BM) that emulates the sequential circuit shown in Fig. 1. First, the combinational circuit is represented by a decision diagram. Next, it is translated into the codes of the BM. Finally, these codes are executed by the BM. To emulate the sequential circuit, the BM uses registers that store state variables. In this section, first, we introduce multi-terminal binary decision diagrams (MTBDDs) that represent multi-output logic functions. Next, we show instructions that evaluate MTBDDs. Then, we show an instruction that reduces the computation time and the code size. Finally, we show the architecture of the BM.

We assume that the BM uses 32-bit instructions, which match the data size of embedded systems and the embedded memory of FPGAs.

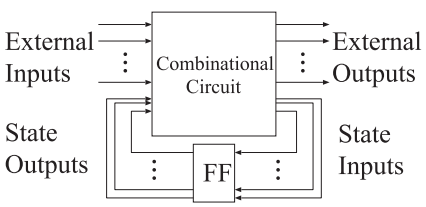


**Fig. 1** Model for a sequential circuit.

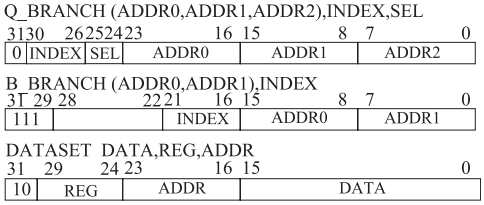


**Fig. 2** Mnemonics and internal representations.

Copyright © 2010 The Institute of Electronics, Information and Communication Engineers

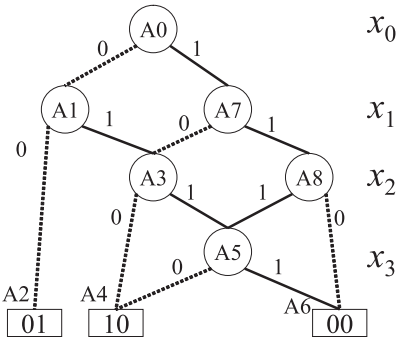


**Fig. 3** Example of MTBDD.

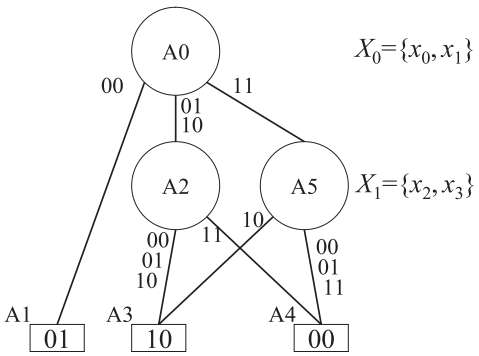


**Fig. 4** MTQDD derived from MTBDD in Fig. 3.

```
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. 5** Program code for the MTBDD in Fig. 3.

```
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. 6** Program code for the MTQDD in Fig. 8.

## 2.1 MTBDD and MTQDD

An arbitrary $n$-variable logic function can be represented by a BDD (Binary Decision Diagram) [4]. An MTBDD (Multi-Terminal Binary Decision Diagram) [13] can evaluate many outputs at a time. Evaluation of an MTBDD requires $n$ table look-ups. The APL (average path length) of a BDD denotes the average number of nodes to traverse the BDD. Evaluation time for a BDD is proportional to the APL [5].

To further speed up the evaluation, an MDD (Multi-valued Decision Diagram) [14] is used. In the MDD ($k$), $k$ variables are grouped to form a $2^k$-valued super variable. Note that a BDD is equivalent to an MDD (1). When the function is represented by an MDD ($k$), at most $\lceil \frac{n}{k} \rceil$ table look-ups are necessary to evaluate an input vector [12]. The evaluation time can be reduced by increasing $k$. However, a node for MDD ($k$) requires pointers proportional to $2^k$. For many benchmark functions, total memory size for MDD ($k$) achieves its minimum when $k = 2$ [15]. Hence, in logic evaluation, with regard to the area-time complexity, MDD (2)s are more suitable than BDDs. Since MDD (2) has 4 branches, it is denoted by a QDD (Quaternary Decision Diagram). In the MDD, we assume that the number of binary variables in the groups can be different. Such MDDs are heterogeneous MDDs. When the groups have the same numbers of binary variables, the MDD is a homogeneous MDD. In this paper, a QDD denotes a heterogeneous decision diagram where each group has either one or two binary variables.

**Example 2.1:** Figure 3 shows an example of MTBDD. Figure 4 shows the MTQDD that is derived from the MTBDD in Fig. 3. (End of Example)

## 2.2 Instructions to Evaluate MTQDDs

Three types of instructions are used to evaluate an MTQDD. A 2-address binary branch instruction (B_BRANCH) and a 3-address quaternary branch instruction (Q_BRANCH)

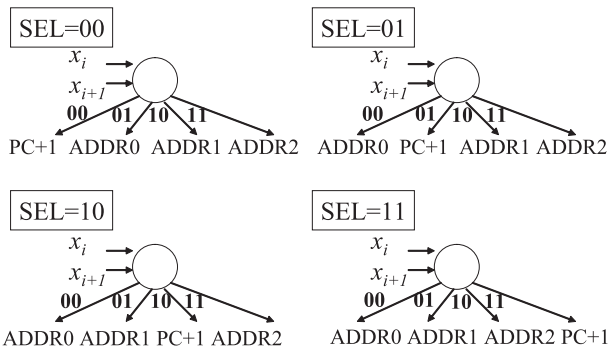


**Fig. 7** Four different Q_BRANCH instructions

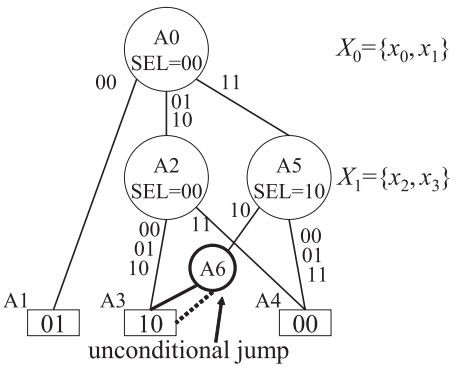


**Fig. 8** MTQDD with 3-address quaternary branch instructions.

evaluate a non-terminal node, while a dataset instruction (DATASET) evaluates a terminal node [21]. Mnemonics and their internal representations for *B_BRANCH*, *Q_BRANCH* and *DATASET* are shown in Fig. 2.

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

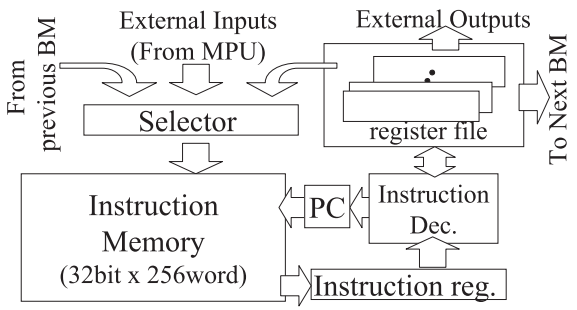


**Fig. 9** BM for a sequential circuit.

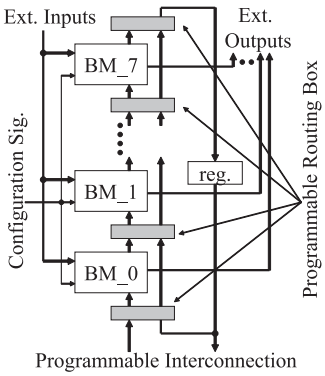


**Fig. 10** Architecture of 8_BM.

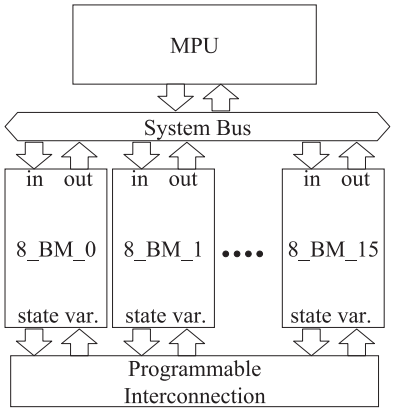


**Fig. 11** Parallel branching program machine (PBM128).

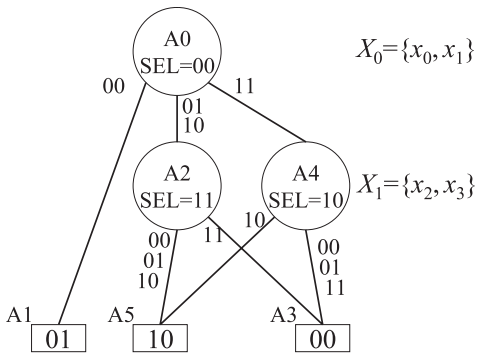


**Fig. 12** An optimal assignment of Fig. 8.

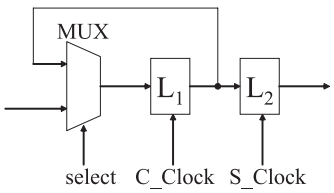


**Fig. 13** Double-rank flip-flop.

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. 7. *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 following example illustrates this:

**Example 2.2:** The program in Fig. 5 evaluates the MTBDD in Fig. 3. Consider the MTQDD shown in Fig. 4. Figure 8 shows the MTQDD with address assignment for *Q_BRANCH* instructions, where *SEL* has the same meaning as Fig. 7. For A6, *B_BRANCH* instruction is used for an unconditional jump, since a terminal node *'10'* is already assigned as A3. Thus, the program in Fig. 6 evaluates the MTQDD. (End of Example)

By changing the address and the $SEL$ as shown in Fig. 12, we can remove the unconditional jump. In this way, for a 3-address quarternary branch, we can optimize the code. The number of unconditional jumps can be minimized by an optimization method shown in [21].

### 2.3 Branching Program Machine for a Sequential Circuit

Figure 9 shows a branching program machine (BM) for a sequential circuit. 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. A Double-Rank Filp-Flop is used to implement the state register and the output register [20]. Figure 13 shows the Double-Rank Filp-Flop, where $L_1$ and $L_2$ are D-latches. The *DATASET* instruction sends the values 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.

In the BM, values of the state register are fed back into its inputs. Thus, the BM can emulate a sequential circuit. A BM can load the external inputs, the state variables, and the outputs from other BMs by specifying the value of the input select register.

## 3. Parallel Branching Program Machine

Since the combinational part of a sequential circuit usually has many inputs and outputs, a direct implementation by a single QDD is too large. Also, the QDD with many outputs tend to have large APL. We partition the outputs of the combinational part into groups, and realize a QDD for each of them. We can emulate them by the parallel branching machine (PBM). In this paper, the PBM$n$ consists of $n$ BMs.

### 3.1 8_BM

Figure 10 shows the architecture of the 8_BM consisting of eight BMs. The output registers of BMs are connected in cascade through programmable routing boxes. Then, these values are stored into the common registers of the 8_BM. Also, the values of registers are fed back to the input of $BM_0$. 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. 10), 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 a 8_BM. Since a BM uses two clocks to perform an instruction, the communication delay within an 8_BM can be neglected.

### 3.2 Optimum Number of BMs

In this part, we consider the optimum number of BMs. For several benchmark functions [24], we partition the outputs into groups using a partition method [17], and constructed the QDD that realizes each group. Then, we obtained the maximum number of nodes and the maximum APL in groups.

Figure14 shows the maximum APL for the benchmark functions, for different numbers of groups $n_{GRP}$ = 32, 64, 128, and 256. From Fig. 14, we have the following:

1. The maximum APL for $n_{GRP}$ = 64 is, on the average, 70.1% of that for $n_{GRP}$ = 32.
2. The maximum APL for $n_{GRP}$ = 128 is, on the average, 80.9% of that for $n_{GRP}$ = 64.
3. However, the maximum APL for $n_{GRP}$ = 256 is, on the average, 98.3% of that for $n_{GRP}$ = 128. This is because, for $n_{GRP}$ = 128, the QDDs with large APLs are already partitioned to singe QDDs, and we cannot partition them any more.

We assume that each group is evaluated by a BM in parallel. Since the evaluation time on BMs is dominant, the connection time for BMs is negligible. Thus, the execution time for multiple BMs is proportional to the maximum APL. From the above observations, the PBM64 ($n_{GRP}$ = 64) is 1.42 times faster than the PBM32 ($n_{GRP}$ = 32); the PBM128 ($n_{GRP}$ = 128) is 1.23 times faster than the PBM64. However, the PBM256 ($n_{GRP}$ = 128) is only 1.01 times faster than the PBM128.

Figure 15 shows the maximum number of nodes for the benchmark functions. In this research, since the number of steps of a BM is 256, it can realize the QDD that has less than or equal to 256 nodes. Figure 15 shows that, when $N_{GRP}$ = 128 and $N_{GRP}$ = 256, all benchmark functions satisfy the restriction.

From the above discussion, we selected the parallel

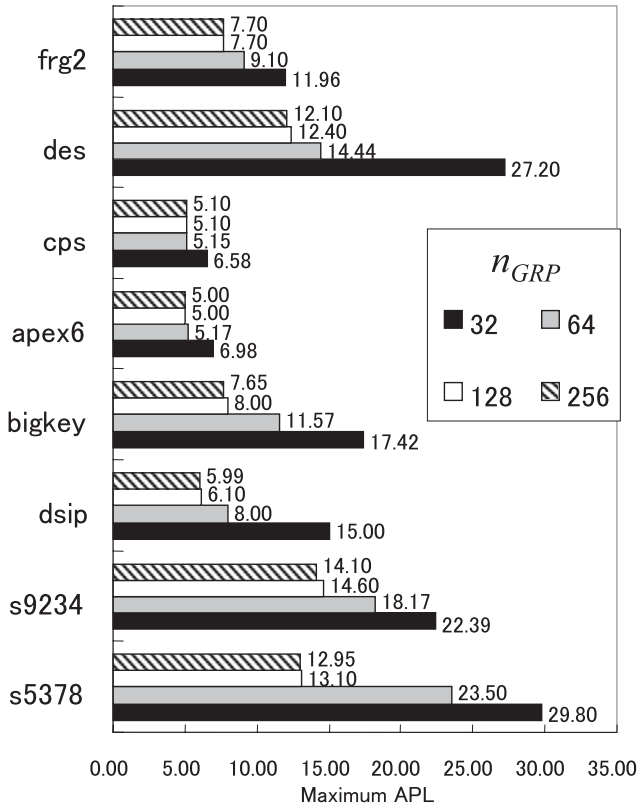


**Fig. 14** Maximum APL vs. number of BMs.

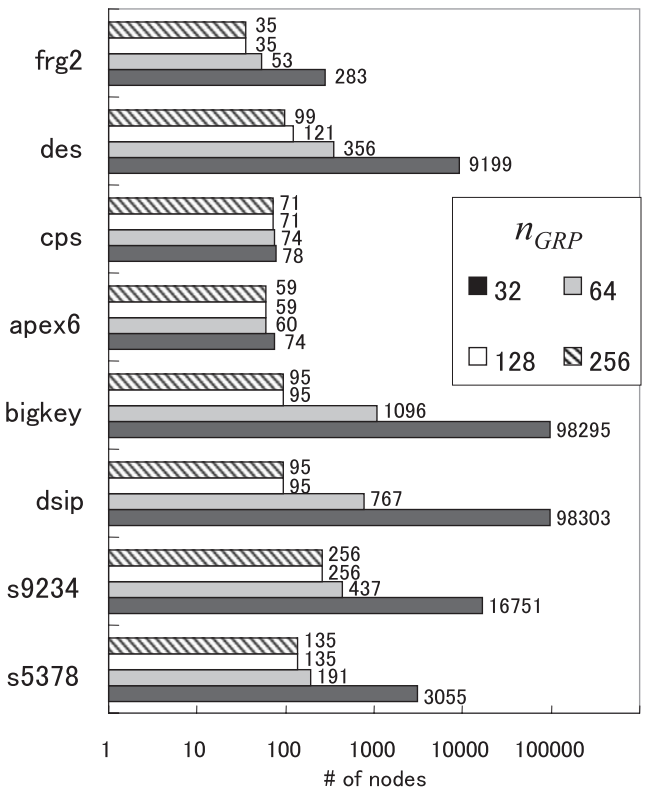


**Fig. 15** Maximum number of nodes vs. number of BMs.

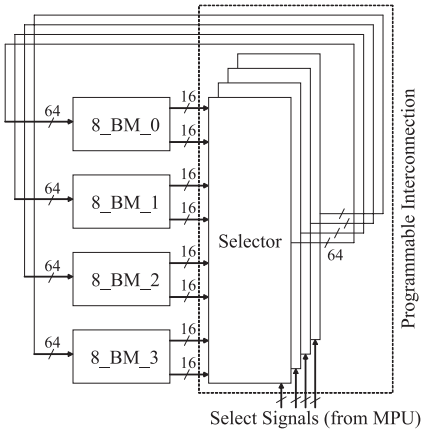


**Fig. 16** An example of the programmable interconnection.

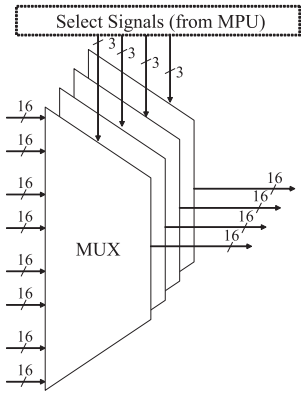


**Fig. 17** Selector for the programmable interconnection.

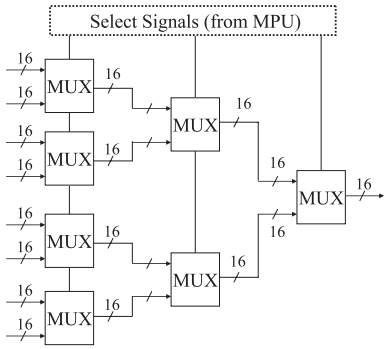


**Fig. 18** Multiplexer for the programmable interconnection.

branching program machine consisting of 128 BMs.

### 3.3 Parallel Branching Program Machine

Figure 11 shows the PBM128. Eight BMs constitute an 8_BM, and sixteen 8_BMs and a programmable interconnection constitute the PBM128. Primary inputs and configuration signals are sent to the 8_BMs. Each 8_BM has external outputs and state variables. The external outputs are connected to the system bus, while the state variables are sent to 8_BMs through the programmable interconnection. In addition, an MPU is used to control the whole system.

### 3.4 Programmable Interconnection

The programmable interconnection uses a multi-level circuit of multiplexers. Figure 16 shows an example of the programmable interconnection. Note that, we use the PBM32 rather than the PBM128 for illustration. To reduce the number of select signals, the programmable interconnection selects four outputs (16 bits) from 8_BMs. Since the *DATASET* can writes 16 bits data in one instruction, it is easy to connect the 8_BMs. Figure 17 shows the selector for the programmable interconnection. It selects four signals (16 bits) from eight inputs (16 bits) using 16 bits multiplexers (MUX) shown in Fig. 18.

To increase the throughput, pipeline registers are inserted into the programmable interconnection. The insertion of pipeline registers increases the latency: Four clocks are used to connect the outputs of an 8_BM to other 8_BM. Since two clocks are used for an instruction of the BM, the PBM128 requires two instructions time to finish the connection between BMs in different 8_BMs. In the code generation, the wait instruction (NOP) is inserted. The unconditional jump instruction is substituted for the NOP instruction.

## 4. Experimental Results and Analysis

### 4.1 Implementation of PBM128

We implemented the PBM128 on Terasic Corp. DE3 development board that contains an FPGA (StratixIII: EP3S340H1152C4)†. For the FPGA synthesis tool, we used QuartusII (v.8.0). To control the whole system, to send and receive the data, and to configure the PBM128, the embedded processor NiosII/e on the FPGA is used. To store the configuration data, the SD-Card to the FPGA board is used. The implemented SD-Card controller reads the configuration data from the external SD-Card. The PBM128 consumes 63007 ALUTs out of 270400 available ALUTs. Each BM consumes 455 ALUTs, each 8_BM consumes 3560 ALUTs, and the programmable interconnection consumes 6046 ALUTs. Also, the PBM128 consumes 128 M9ks out of 1040 available M9ks. In our implementation, the maximum frequency was 132.73 [MHz].

### 4.2 Comparison of the Code Size and the CPU Time

We compared the execution time and code size for the PBM128 with the Intel's general-purpose processor Core2Duo using the benchmark functions [24]. We used a laptop computer using Intel's Core2Duo U7600 (1.2 GHz, Cache L1 data 32 KB, L1 instruction 32 KB, and L2 2 MB), and OS: Windows XP SP2. The execution code was generated by gcc compiler with optimization option -O3††. The numbers of inputs and outputs for the selected benchmark functions are too large to be represented by a single MTQDD. Thus, we partitioned the outputs into groups, then represented them by multiple MTQDDs, and finally converted them into the codes for the PBM128. We used a grouping method that partitions outputs with similar inputs.

†This device contains more ALUTs than that of [18].

††The table look-up method [19] can also evaluate the BDD. However, for selected benchmark functions, it is slower than our method.

**Table 1** Comparison of execution code size and execution time.

| Name | In | Out | FF | | Core2Duo@1.2 GHz | | | | PBM128@100 MHz | | | | Ratios (Core2Duo/PBM128) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Total Grp | BNode | Total APL | Code [KB] | Time [ns] | QNode with UJ | Max. APL | Code [KB] | Time [ns] | Ratio.Code | | Ratio.Time | |
| | | | | | | | | | | | | | Est. | Act. | Est. | Act. |
| s5378 | 35 | 49 | 164 | 126 | 5702 | 703.9 | 74.6 | 12030 | 4131 | 13.1 | 17.8 | 323 | 4.14 | 4.19 | 49.1 | 37.2 |
| s9234 | 36 | 39 | 211 | 117 | 10963 | 590.7 | 148.6 | 13450 | 7613 | 14.6 | 33.4 | 352 | 4.32 | 4.44 | 46.9 | 38.2 |
| dsip | 229 | 197 | 224 | 120 | 7907 | 649.3 | 112.1 | 17500 | 5342 | 6.1 | 24.8 | 182 | 4.44 | 4.52 | 95.0 | 96.1 |
| bigkey | 263 | 197 | 224 | 125 | 9971 | 831.7 | 149.5 | 19170 | 6876 | 8.0 | 33.9 | 220 | 4.35 | 4.41 | 98.9 | 87.1 |
| apex6 | 135 | 99 | | 99 | 1535 | 297.1 | 23.0 | 3700 | 1016 | 5.0 | 4.8 | 163 | 4.53 | 4.79 | 21.8 | 22.6 |
| cps | 24 | 102 | | 102 | 3035 | 242.5 | 33.9 | 3468 | 2121 | 5.1 | 8.3 | 162 | 4.29 | 4.08 | 19.1 | 21.4 |
| des | 256 | 245 | | 124 | 8952 | 770.3 | 123.1 | 16560 | 6730 | 12.4 | 30.7 | 308 | 3.99 | 4.00 | 50.2 | 53.7 |
| frg2 | 143 | 139 | | 116 | 3161 | 529.9 | 40.0 | 6390 | 2226 | 7.7 | 9.2 | 215 | 4.26 | 4.34 | 28.0 | 29.7 |

```
// C-code                    // Assembly-code
A_1:                         A_1:
if( x[2] & 0x001)            movl %eax,_x+8
goto A_2;                    testb %al,$1
else                         je A_10
goto A_10;                   A_2:
A_2:                         movl %eax,_x+4
if( x[1] & 0x002)            testb %al,$2
goto A_4;                    jne A_4
else
goto A_3;
```

**Fig. 19** C-code and assembly-code for a node of a BDD.

```
// C-code                    // Assembly-code
A_1:                         A_1:
switch( x[2] & 0x3){         movl %eax,_x+8
case 0x0:  goto A_2;         andl %eax,$3
break;                       cmpl %eax,$1
case 0x1:  goto A_10;        je A_2
break;                       jb A_10
case 0x2:  goto A_4;         cmpl %eax,$2
break;                       je A_4
case 0x3:goto A_3;           A_3:
}
```

**Fig. 20** C-code and assembly-code for a node of a QDD.

As for the data structure, the MTQDD is used for the PBM128, while the MTBDD is used for the Core2Duo. This is from the following reason. Figure 19 shows the C-code and the assembly-code for the MPU representing a non-terminal node of a BDD, while Fig. 20 shows those of a QDD. To evaluate a non-terminal node for a BDD, the assembly-code shown in Fig. 19 performs the following operations:

1. Read an input variable by *movl* instruction.
2. Extract the specified bit by *testb* instruction.
3. Perform the conditional branch by *je* instruction.

These operations evaluate a node of a BDD using the 1-address branch, and they are emulated by three x86 instructions. On the other hand, for a QDD, the assembly-code shown in Fig. 20 performs the following operations:

1. Read an input variable by *movl* instruction.
2. Extract the specified bit by *andl* instruction.
3. Check a lower bit and set frags *CF, ZF* by *cmpl* instruction.
4. Perform the branch by *je* and *jb* instructions.

To evaluate a non-terminal node, the average number of steps is $\frac{4+5+7+7}{4} = 5.75$. For benchmark functions, $APL_{MTBDD} = 1.00$, while $APL_{MTQDD}(APL_{MDD(2)}) = 0.69$ [16]. Thus, the average number of steps to evaluate a path for the BDD is $3.00 \times APL_{MTBDD} = 3.00$, while that for the QDD is $5.75 \times APL_{MTQDD} = 3.96$. Therefore, in Core2Duo, the MTBDD is faster than the MTQDD.

For each node of the MTBDD, we generated a fragment of program code (i.e., if then else BRANCH instructions). As a result, the code for the MTBDD has a higher cache hit rate than that for the MTQDD. We used the same partitions of the outputs in the Core2Duo and in the PBM128. To obtain the execution time per a vector, we generated random test vectors, and obtained the average time excluding the time for the reading and writing vectors. The frequency for the PBM128 was 100 [MHz], while that for the Core2Duo was 1.2 [GHz]. The code size of the MPU is obtained as the size of the execution code for the MTBDD minus the code size to generate test vectors and to measure the execution time. The code size for the PBM128 is derived from the total number of steps. Table 1 compares the code size and the execution time for the Core2Duo and the PBM128. In Table 1, *Name* denotes the name of benchmark function; *In* denotes the number of inputs; *Out* denotes the number of outputs; *FF* denotes the number of state variables[†]; *Total Grp* denotes the total number of groups; *BNode* denotes the sum of the number of nodes for the MTBDD; *QNode with UJ* denotes the sum of the number of nodes for the MTQDD and the number of unconditional jump instructions; *Code* denotes the size of execution code [KBytes]; *Time* denotes the execution time [nsec]; *Total APL* denotes the sum of the average path length (APL) for all the groups; *Max. APL* denotes the maximum APL among the groups; and *Ratios* denote that for the code size and that of the execution time (Core2Duo/PBM128). Note that, *Est.* denotes the estimated ratio described later, and *Act.* denotes the actual ratio from the experiment. Table 1 shows that the PBM128 requires approximately a quarter of the memory for the Core2Duo, and is 21.4-92.5 times faster than the Core2Duo.

[†]For combinational circuits (apex6, cps, des, and frg2), the numbers of state variables are zero.

### 4.3 Analysis of Execution Code Size

Let $Est.Code.MPU$ be the estimated number of steps in the MPU. Note that, the MPU evaluates MTBDDs. As shown in Fig. 19, three instructions are used to evaluate a node of a MTBDD. Thus, we have

$$Est.Step.MPU = BNode \times 3, \tag{1}$$

where *BNode* is the number of MTBDD nodes in Table 1.

Let $Est.Step.PBM$ be the estimated number of steps in the PBM128. To evaluate MTQDDs on the PBM128, *Q_BRANCH* instructions and *unconditional jump* instructions are necessary. So, we have

$$\begin{aligned} Est.Step.PBM &= QNode + UJ \\ &= QNode\ with\ UJ \end{aligned} \tag{2}$$

where *QNode* is the number of MTQDD nodes, $UJ$ is the number of *unconditional jump* instructions, and *QNode with UJ* is the sum of these values shown in Table 1.

Let $Est.Ratio.Step$ be the estimated ratio for the number of steps. From Exprs. (1) and (2), we have

$$\begin{aligned} Est.Ratio.Step &= \frac{Est.Step.MPU}{Est.Step.PBM} \\ &= \frac{BNode}{QNode\ with\ UJ} \times 3 \end{aligned} \tag{3}$$

From Table 1, we can see that $Est.Ratio.Step$ is 3.99-4.53, while *Act.Ratio.Step* is 4.00-4.79. In short, Core2Duo requires about 4 times more steps than the PBM128.

### 4.4 Analysis of Execution Time

Let $Est.Time$ be the estimated execution time for a decision diagram. Then, we have the following relation:

$$Est.Time = ETPI \times IPN \times APL + TST, \tag{4}$$

where *ETPI* [nsec/inst] denotes the execution time per an instruction; *IPN* [inst/node] denotes the number of instructions per a node; *APL* [node] denotes the average path length; and *TST* [nsec] denotes the time to perform the state transition.

Let *Est.Time.PBM* be the estimated execution time in the PBM128. Since the PBM128 uses two clocks and the operation frequency is 100 [MHz] to execute an instruction, we have $ETPI_{PBM} = 20\,[nsec/inst]$. A *Q_BRANCH* instruction evaluates a node for the MTQDD. Thus, we have $IPN_{PBM} = 1.0$. The PBM128 emulates QDDs in parallel. So, the *APL* of the PBM128 is bounded by the maximum *APL* of all the groups. Let $APL_{PBM} = \max\{QAPL_i\}$, where $QAPL_i$ denotes the APL for the QDD that represents the $i$-th group. The programmable interconnection propagates the state variables in four clocks. Thus, we have $TST_{PBM} = 40\,[nsec]$.

Let *Est.Time.MPU* be the estimated execution time in the general-purpose processor. Note that, $ETPI_{MPU}$ depends on the cache access time. Execution time of an instruction depends on which of the caches is used: $L_1$ or $L_2$. To obtain the access time for $L_1$ and $L_2$ cache, we did the additional experiment: The average access time for $L_1$ cache is about 3 [nsec], while that for $L_2$ cache is about 15 [nsec]. As shown in Fig. 19, to evaluate a non-terminal node for a BDD on the MPU, three instructions (*mov*, *test*, and *jump*) are used. Generally, jump addresses in the *B_BRANCH* instructions are random. Thus, the first instruction (*mov*) causes the cache miss, which makes the instruction slow. On the other hand, other two instructions (*test* and *jump*) are fast, since these instructions are prefetched to the $L_1$ cache. Let $T_{L_1}$ be the access time for the $L_1$ cache; $T_{L_2}$ be the access time for the $L_2$ cache; $M_{L_1}$ be the code size for the $L_1$ cache; $M_{BDD}$ be the code size for the BDD; and $T_{mov}$ be the time for the *mov* instruction that considers the cache miss. Note that, the hit rates of the caches $L_1$ and $L_2$ are $\frac{M_{L_1}}{M_{BDD}}$ and $\frac{M_{BDD}-M_{L_1}}{M_{BDD}}$, respectively. Thus, we have

$$T_{mov} = \frac{T_{L_2}(M_{BDD} - M_{L_1}) + T_{L_1}M_{L_1}}{M_{BDD}}. \tag{5}$$

Also, we have

$$ETPI_{MPU} = \begin{cases} T_{L_1} & (M_{BDD} \le M_{L_1}) \\ \frac{T_{mov}+2\times T_{L_1}}{3} & (M_{BDD} > M_{L_1}). \end{cases} \tag{6}$$

In Expr. (6), the upper expression shows $ETPI_{MPU}$ without cache misses, and the lower one shows $ETPI_{MPU}$ with cache misses. Obviously, when $M_{BDD} \le M_{L_1}$, no cache miss occurs. In the lower expression of Expr. (6), the second term in a numerator denotes the estimated execution time for *test* and *jump* operations. From Fig. 19, we have $IPN_{MPU} = 3.0$. For the *APL* of the MPU, all BDDs are evaluated sequentially. Thus, we have $APL_{MPU} = \sum_{i=0}^{g-1} BAPL_i$. Note that, $BAPL_i$ is the APL for the BDD representing the $i$-th group, and $g$ is the number of groups. For the state transition, the MPU must update the state variables sequentially. We assume that state variables are stored in the $L_1$ cache which can be accessed in 3 [nsec]. So, we have $TST_{MPU} = \#FF \times 3$, where $\#FF$ denotes the number of state variables in Table 1.

Let *Est.Ratio.Time* be the estimated ratio for the execution time. Then, we have

$$Est.Ratio.Time = \frac{Est.Time.MPU}{Est.Time.PBM}. \tag{7}$$

From Table 1, we can see that *Est.Ratio.Time* is 19.1-98.9, while *Act.Ratio.Time* is 21.4-96.1.

From the observations, the PBM128 is faster than the MPU by:

1. Eliminating the cache miss,
2. Using a special *Q_BRANCH* instruction, and
3. Using 128 BMs in parallel.

The Core2Duo requires more memory than the PBM128. So, the cache miss occurs frequently. On the

**Table 2** Comparison of power consumption (W).

| Name | Core2Duo@1.2 GHz | PBM128 |
|---|---|---|
| s5378 | 13.70 | 3.24 |
| s9234 | 13.66 | 3.31 |
| dsip | 13.06 | 3.23 |
| bigkey | 13.68 | 3.21 |
| apex6 | 13.61 | 3.22 |
| cps | 13.21 | 3.29 |
| des | 13.70 | 3.28 |
| frg2 | 13.81 | 3.22 |

other hand, in the PBM128, each BM stores the necessary data in its local memory. Thus, no cache miss occurs. Note that, for the Core2Duo, since it operates on Windows XP, the overhead of the operation system degrades the performance. However, we ignore it.

### 4.5 Comparison of Power Consumption

Table 1 shows that the PBM128 is 21.4-96.1 times faster than the Core2Duo. Even if the clock frequency for the PBM128 is reduced to $\frac{1}{21.4} \sim \frac{1}{96.1}$, the throughput of the PBM128 is the same as the Core2Duo. Here, we consider power consumption. The total power consumption consists of the dynamic power and the static power. To reduce the dynamic power, we reduced the clock frequency for the PBM128 so that the throughput is equal to that of the Core2Duo. On the other hand, the clock frequency of the Core2Duo is kept to 1.2 GHz. Then, we measured the total power consumption. Table 2 compares the total power consumption of the PBM128 and the Core2Duo. To obtain the pure power consumption for the Core2Duo, we turned off the display, and suspended applications except for the kernel and the clock counter for measurement of the execution time. Also, to make the comparison fair, we tried to make the temperature of the PBM128 and the Core2Duo same.

Table 2 shows that the total power consumption for the PBM128 is 23.6% of that for the Core2Duo. The static power consumption for the PBM128 was 3.14 W that comes from the leakage power of the FPGA.

Next, we obtained the relationship between the clock frequency and the power consumption for the PBM128. Let $h$ be the clock frequency (MHz), and $P$ be the total power consumption (W). We measured power consumption several times by changing the clock frequency. By applying the linear approximation, we obtained the following relation:

$$P = 0.0059h + 3.3, \tag{8}$$

where the first term corresponds to the dynamic power, and the second term corresponds to the static power. In Expr (8), when $h = 0$, we have $P = 3.3$ W, that does not match the experimental value (3.14 W). This is because the increase of temperature by the increase of frequency made an approximation error. Expr (8) shows that, in the PBM128, the static power dominates the total power consumption. Thus, the reduction of the chip area can reduce total power consumption. Note that, in the PBM128, we obtained the static power, since we can stop the system clock.

On the other hand, in the Core2Duo, we could not stop the system clock, since dynamic RAM was used. So, we obtained the *standby* power instead of the *static* power. The *Standby* power of the Core2Duo consists of the leakage power, the refresh power of the DRAM, and the power consumption for the kernel. In our experiment, the standby power for the Core2Duo was 8.56 W.

## 5. Implementation of Packet Filter on the PBM128

We implemented an access controller (*acl*) and a firewall (*fw*) for the Internet generated by ClassBench [23]. We used ClassBench to produce synthetic filter sets modeling real filter sets. Then, we compared their memory size and computation time for the PBM128 with ones implemented on the Core2Duo.[†]. ClassBench generates packet filters consisting of rules. A rule consists of six fields: a destination IP address, a source IP address, a destination port, a source port, a protocol number, and a flag[††]. In the packet filter, an entry of the IP address field denotes IP addresses that are detected by longest prefix (LPM) matching; an entry of the port field denotes a range of port numbers that is detected by range matching; and entries of the protocol field and the flag field that denote a protocol and a flag those are detected by exact matching. In a general packet filter, rules may have intersections. However, in this implementation, we assume that rules have no intersection[†††].

**Example 5.3:** Figure 21 shows an example of a packet filter. For simplicity, only three fields are shown: the source IP address, the source port, and the protocol. (End of Example)

Since the packet filter contains range matching, a BDD representing the packet filter becomes too large. So, the direct realization of the packet filter on the PBM128 is difficult. Thus, we use DCFL (Distributed Crossproducting of Field Labels) method [22]. First, we partitioned the packet filter into six fields, and assigned a unique label to each entry of the field. Next, we generated the product table that contains the combinations of fields corresponding to rules for the packet filter. To detect a rule, we used six tables for the fields and the product table.

**Example 5.4:** Figure 22 shows the tables that convert

[†]Different users require systems with different performance. Thus, different architecture should be used. For 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, for low-end users including SOHO (small office and home office), the required throughput is at most several giga bits per second. Thus, 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.

[††]Since the access controller does not use the flag field, we use five fields.

[†††]By using an option '-b', we can generate rules without intersections.

| Rule | Address | Port | Protocol |
|---|---|---|---|
| 1 | 1000 | [0:1] | TCP |
| 2 | 00** | [1:1] | TCP |
| 3 | 010* | [5:15] | UDP |
| 4 | 0*** | [3:8] | TCP |
| 5 | 10** | [9:15] | TCP |

**Fig. 21** An example of packet filter.

**Table 3** Comparison the PBM128 with the Core2Duo for the packet filter.

| Rule | | Core2Duo@1.2 GHz | | | PBM128@100 MHz | | |
|---|---|---|---|---|---|---|---|
| | Total Grp | Total APL | Code [KB] | Time [ns] | Max. APL | Code [KB] | Time [ns] |
| acl | 86 | 187.0 | 157.6 | 1683.8 | 3.6 | 27.0 | 78.8 |
| fw | 99 | 190.8 | 160.7 | 1717.3 | 3.3 | 27.9 | 72.3 |

| Address | Label |
|---|---|
| 1000 | 1 |
| 00** | 2 |
| 010* | 3 |
| 011* | 4 |
| 1001 | 5 |
| 101* | 5 |
| 11** | 5 |

| Port | Label |
|---|---|
| [0:0] | 1 |
| [1:1] | 2 |
| [2:2] | 3 |
| [3:4] | 4 |
| [5:8] | 5 |
| [9:15] | 6 |

| Protocol | Label |
|---|---|
| TCP | 1 |
| UDP | 2 |

**Fig. 22** Tables for fields.

fields into labels. They correspond to the packet filter shown in Fig. 21. Figure 24 shows the product table for Fig. 22. (End of Example)

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*'. Next, we partitioned the rules into 10 subsets, each consisting of 20 rules (Fig. 23 Step 1). Then, we partitioned each subset into 6 fields (Fig. 23 Step 2), and generated the product table (Fig. 23 Step 3). Next, we constructed seven BDDs corresponding to six field tables and the product table (Fig. 23 Step 4). Finally, we partition the BDD by each bit of outputs (Fig. 23 Step 5), and converted to many QDDs. We stored the program code for generated QDDs into the PBM128. In the Core2Duo, the code for the BDD is simpler than that for the QDD, thus, the code for the BDD has a higher cache hit rate than that for the QDD. So, the Core2Duo emulates BDDs instead of QDDs. We used the same partitions of the outputs in the Core2Duo and in the PBM128. To obtain the execution time per a vector, we generated random packet headers, and obtained the average time excluding the time for the reading and writing packet headers.

Table 3 compares memory size and computation time, where column labels are the same as Table 1. From Table 3, we can observe that, as for memory size, the PBM128 requires 17.1%-17.3% of the memory for the Core2Duo, and as for the speed, the PBM128 is 21.3-23.7 times faster than the Core2Duo.

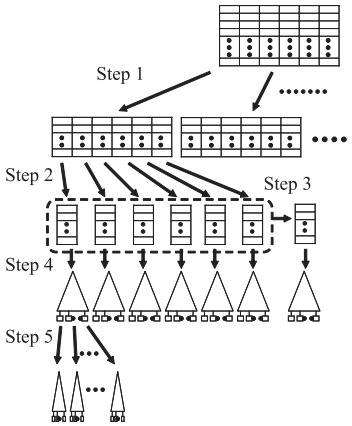


**Fig. 23** Realization of packet filter.

| Combination | Rule |
|---|---|
| (1,1,1) | 1 |
| (1,2,1) | 1 |
| (2,2,1) | 2 |
| (3,5,2) | 3 |
| (3,6,2) | 3 |
| (2,4,1) | 4 |
| (2,5,1) | 4 |
| (3,4,1) | 4 |
| (3,5,1) | 4 |
| (4,4,1) | 4 |
| (4,5,1) | 4 |
| (1,6,1) | 5 |
| (5,6,1) | 5 |

**Fig. 24** The product table for Table 3.

## 6. Conclusion

In this paper, we presented the PBM128 that consists of 128 BMs and a programmable interconnection. To represent logic functions on BMs, we used quaternary decision diagrams. To evaluate functions, we used 3-address quaternary branch instructions. We emulated many benchmark functions on the PBM128 and the Intel's Core2Duo microprocessor. The PBM128 required approximately a quarter of the memory of the Core2Duo, was 21.4-96.1 times faster than the Core2Duo, and dissipated a quarter of the power of the Core2Duo.

Three tricks for the fast operation are:

1. Special conditional branch instructions that evaluate two variables at a time.
2. Parallel operation of 128 BMs.
3. Elimination of cache misses by distributed memories and by special instructions that reduce the memory size.

Also, we implemented two types of packet filters; the access controller and the fire wall. For these applications, the PBM128 requires approximately 17% of the memory for the Core2Duo, and is 21.3-23.7 times faster than the Core2Duo.

## Acknowledgments

This research is supported in part by the Grants in Aid for Scientific Research of JSPS, and the grant of Innovative Cluster Project of MEXT (the second stage). Discussion with Mr.Hisashi Kajiwara was quite useful. Reviewer's comments significantly improved the paper.

[†]With the increase of the number of rules, it causes an unexpected intersection of rules, and decreases the system performance [9]. In the embedded packet filter using the general purpose processor [6], the number of recommended rules is 100-300 [10]. Thus, we set the number of rules to 200.

## References

[1] P. Ashar and S. Malik, "Fast functional simulation using branching programs," Proc. International Conference on Computer Aided Design, pp.408–412, Nov. 1995.
[2] P.C. Baracos, R.D. Hudson, L.J. Vroomen, and P.J.A. Zsombor-Murray, "Advances in binary decision based programmable controllers," IEEE Trans. Ind. Electron., vol.35, no.3, pp.417–425, Aug. 1988.
[3] R.T. Boute, "The binary-decision machine as programmable controller," Euromicro Newsletter, vol.1, no.2, pp.16–22, 1976.
[4] R.E. Bryant, "Graph-based algorithms for boolean function manipulation," IEEE Trans. Comput., vol.C-35, no.8, pp.677–691, Aug. 1986.
[5] J.T. Butler, T. Sasao, and M. Matsuura, "Average path length of binary decision diagrams," IEEE Trans. Comput., vol.54, no.9, pp.1041–1053, Sept. 2005.
[6] CISCO: ASA5500 series, http://www.cisco.com/
[7] C.H. Clare, Designing Logic Systems Using State Machines, McGraw-Hill, New York, 1973.
[8] M. Davio, J.-P Deschamps, and A. Thayse, Digital Systems with Algorithm Implementation, p.368, John Wiley & Sons, New York, 1983.
[9] K. Golnabi, R.K. Min, L. Khan, and E. Al-Shaer, "Analysis of firewall policy rules using data mining techniques," 10th IEEE/IFIP Network Operations and Management Symposium (NOMS2006), pp.305–315, April 2006.
[10] P. Gupta and N. McKeown, "Packet classification on multiple fields," ACM Sigcomm, Aug. 1999.
[11] Y. Iguchi, T. Sasao, M. Matsuura, and A. Iseno, "A hardware simulation engine based on decision diagrams," Asia and South Pacific Design Automation Conference'2000, pp.73–76, Yokohama, Japan, Jan. 2000.
[12] Y. Iguchi, T. Sasao, and M. Matsuura, "Implementation of multiple-output functions using PROMDDs," 30th International Symposium on Multiple-Valued Logic, pp.199–205, Portland, Oregon, U.S.A, May 2000.
[13] Y. Iguchi, T. Sasao, and M. Matsuura, "Evaluation of multiple-output logic functions," Asia and South Pacific Design Automation Conference'2003, pp.312–315, Kitakyushu, Japan, Jan. 2003.
[14] 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.
[15] S. Nagayama, T. Sasao, Y. Iguchi, and M. Matsuura, "Area-time complexities of multi-valued decision diagrams," IEICE Trans. Fundamentals, vol.E87-A, no.5, pp.1020–1028, May 2004.
[16] S. Nagayama and T. Sasao, "Compact representations of logic functions using heterogeneous MDDs," IEICE Trans. Fundamentals, vol.E86-A, no.12, pp.3168–3175, Dec. 2003.
[17] H. Nakahara, T. Sasao, and M. Matsuura, "A design algorithm for sequential circuits using LUT rings," IEICE Trans. Fundamentals, vol.E88-A, no.12, pp.3342–3350, Dec. 2005.
[18] H. Nakahara, T. Sasao, K. Matsuura, and Y. Kawamura, "Emulation of sequential circuits by a parallel branching program machine," 5th International Workshop on Applied Reconfigurable Computing (ARC'09), Karlsruhe, Germany, March 2009, Lect. Notes Comput. Sci., LNCS5443, pp.261–267, March 2009.
[19] P.C. McGeer, K.L. McMillan, A. Saldanha, A.L. Sangiovanni-Vincentelli, and P. Scaglia, "Fast discrete function evaluation using decision diagrams," Proc. International Conference on Computer Aided Design, pp.402–407, Nov. 1995.
[20] T. Sasao, H. Nakahara, M. Matsuura, and Y. Iguchi, "Realization of sequential circuits by look-up table ring," 2004 IEEE International Midwest Symposium on Circuits and Systems, pp.I:517–I:520, Hiroshima, July 2004.
[21] T. Sasao, H. Nakahara, M. Matsuura, Y. Kawamura, and J.T. Butler, "A quaternary decision diagram machine and the optimization of its code," 39th International Symposium on Multiple-Valued Logic (ISMVL 2009), pp.362–369, May 2009.
[22] D.E. Taylor, "Survey and taxonomy of packet classification techniques," ACM Comput. Surv., vol.37, no.3, pp.238–275, Sept. 2005.
[23] D.E. Taylor and J.S. Turner, "ClassBench: A packet classification benchmark," 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2005), vol.3, pp.2068–2079, March 2005.
[24] S. Yang, "Logic synthesis and optimization benchmark user guide version 3.0," MCNC, Jan. 1991.
[25] 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.

**Hiroki Nakahara** received the BE, ME, and Ph.D. degrees in computer science from Kyushu Institute of Technology, Fukuoka, Japan, in 2003, 2005, and 2007, respectively. He has a research position at Kyushu Institute of Technology, Iizuka, Japan. His research interests include logic synthesis, reconfigurable architecture, embedded system, and high-level synthesis. He is a member of the IEEE.

**Tsutomu Sasao** received the BE, ME, and Ph.D. degrees in electronics engineering from Osaka University, Osaka, Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan, the IBM T.J. Watson Research Center, Yorktown Heights, New York, and the Naval Postgraduate School, Monterey, California. He is now a Professor of the Department of Computer Science and Electronics at the Kyushu Institute of Technology, Iizuka, Japan. His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than nine books on logic design, including Logic Synthesis and Optimization, Representation of Discrete Functions, Switching Theory for Logic Synthesis, and Logic Synthesis and Verification, Kluwer Academic Publishers, 1993, 1996, 1999, and 2001, respectively. He has served as Program Chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. Also, he was the Symposium Chairman of the 28th ISMVL held in Fukuoka, Japan, in 1998. He received the NIWA Memorial Award in 1979, Distinctive Contribution Awards from the IEEE Computer Society MVL-TC for papers presented at ISMVLs in 1986, 1996, 2003 and 2004, and Takeda Techno-Entrepreneurship Award in 2001. He has served as an Associate Editor of the IEEE Transactions on Computers. He is a fellow of the IEEE.

**Munehiro Matsuura** studied at the Kyushu Institute of Technology from 1983 to 1989. He received the B.E. degree in Natural Sciences from the University of the Air, in Japan, 2003. He has been working as a Technical Assistant at the Kyushu Institute of Technology since 1991. He has implemented several logic design algorithms under the direction of Professor Tsutomu Sasao. His interests include decision diagrams and exclusive-OR based circuit design.

**Yoshifumi Kawamura** graduated from Electrical Engineering of Miyagi Technical College. In 1981, he entered the Semiconductor Division of Hitachi, Ltd., and engaged in the development of Telecommunication devices. In 1993, he transferred Graphics Communication Laboratories, and engaged in the research and development of MPEG2 System for four years. In 1997, he transferred to the Semiconductor Div. Hitachi Ltd., and engaged in development of LSI for GSM Cell phone. In 2003, he transferred to the Corporate Strategy Planning Division, Renesas Technology Corporation. Now, he is a senior engineer of the System Core Technology Division. He is involved in the development of programmable and reconfigurable technology.