Code Generation for Embedded Systems Using Heterogeneous MDDs

Shinobu Nagayama[1]

[1]Department of Computer Science and Electronics
Kyushu Institute of Technology
Iizuka, Japan 820-8502
nagayama@aries02.cse.kyutech.ac.jp

Tsutomu Sasao[1,2]

[2]Center for Microelectronics System
Kyushu Institute of Technology
Iizuka, Japan 820-8502
sasao@cse.kyutech.ac.jp

Abstract— This paper proposes a method to generate program code for embedded systems using Multi-valued Decision Diagrams (MDDs) that are called heterogeneous MDDs. The heterogeneous MDDs represent logic functions more compactly, and have shorter average path length than other Decision Diagrams (DDs). The code generated using heterogeneous MDDs can evaluate logic functions faster using a small amount of memory. Our experimental results show that some functions are suitable for this method, while others are suitable for Levelized Compiled Code (LCC) method. We also introduce a new measure of logic functions, the LN-ratio to determine which of the two methods is better.

I INTRODUCTION

Embedded systems are widely used in vehicle control, consumer electronics, Personal Digital Assistance (PDA), cellular phone, and so on. These embedded systems are usually composed of MPUs and software programs to reduce time-to-market. These software programs have the restriction on the amount of memory to reduce their cost, power consumption, weight, and so on. Therefore, synthesis of efficient software programs with limited code size is important.

Synthesis of software programs is the automatic generation of a program from a functional specification such as a Finite State Machine (FSM). Software synthesis using Binary Decision Diagrams (BDDs) [2], Free BDDs (FBDDs) [12], and multi-valued logic network [27] have been proposed. These approaches generate code using Decision Diagrams (DDs). In this paper, we propose a method to generate efficient program code using heterogeneous Multi-valued Decision Diagrams (MDDs).

The rest of the paper is organized as follows. In Section II, we define heterogeneous MDDs and a method to represent multiple-output functions. In Section III, we introduce the average path length to estimate the computation time. In Section IV, we propose a method to generate the optimal code using heterogeneous MDDs. And, in Section V, we compare the performance of code generated by three different methods. Also, we introduce a new measure of the logic function, the LN-ratio, which shows a suitable code generation method.

II DEFINITIONS AND BASIC PROPERTIES OF MDDS

This section defines heterogeneous MDDs, and shows a method to represent multiple-output functions.

A Representation of Logic Functions

Let $f(X)$ be a two-valued logic function, where $X = (x_1, x_2, \ldots, x_n)$. Let $x_i (i = 1, 2, \ldots, n)$ be binary variables.

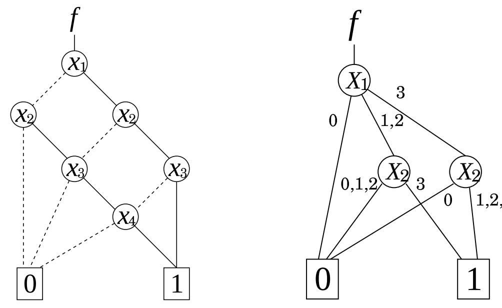
Let $\{X\}$ denote the set of variables in X. If $\{X\} = \{X_1\} \cup \{X_2\} \cup \ldots \cup \{X_u\}$ and $\{X_i\} \cap \{X_j\} = \phi, i \neq j$, then $(X_1, X_2, \ldots, X_u)$ is a **partition** of X. An ordered set of variables X_i is called a **super variable**. If $|X_i| = k_i$ $(i = 1, 2, \ldots, u)$ and $k_1 + k_2 + \ldots + k_u = n$, then a two-valued logic function $f(X)$ can be represented by the mapping $f(X_1, X_2, \ldots, X_u)$: $P_1 \times P_2 \times P_3 \times \ldots \times P_u \rightarrow B$, where $P_i = \{0, 1, 2, \ldots, 2^{k_i} - 1\}$ and $B = \{0, 1\}$. We assume that the function is completely specified and includes no redundant variables.

Example 1 *Consider (X_1, X_2) which is a partition of X, where $X = (x_1, x_2, x_3, x_4, x_5)$ and each x_i is a binary variable. When $X_1 = (x_1, x_2)$ and $X_2 = (x_3, x_4, x_5)$, $k_1 = 2$, $k_2 = 3$, $P_1 = \{0, 1, 2, 3\}$, and $P_2 = \{0, 1, \ldots, 7\}$. Note that X_1 takes 4 values, and X_2 takes 8 values. So, a 5-variable logic function $f(X)$ can be represented by the multi-valued function $f(X_1, X_2)$: $P_1 \times P_2 \rightarrow B$. (End of Example)*

B Heterogeneous MDD

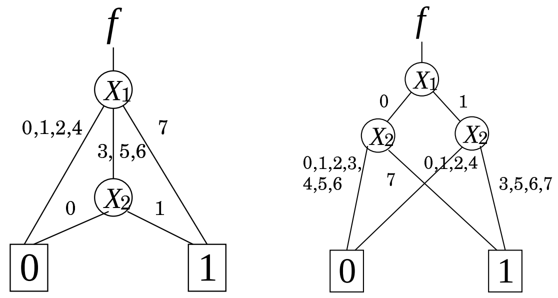
We assume that readers are familiar with BDDs, Reduced Ordered BDDs (ROBDDs) [6], MDDs, and Reduced Ordered MDDs (ROMDDs) [11].

Definition 1 *When $X = (x_1, x_2, \ldots, x_n)$ is partitioned into $(X_1, X_2, \ldots, X_u)$, an ROMDD representing a logic function $f(X)$ is called a* **heterogeneous MDD**. *Specifically, when $k_1 = k_2 = \ldots = k_u$, an ROMDD representing a logic function $f(X)$ is called a* **homogeneous MDD** *in order to distinguish from a heterogeneous MDD. A homogeneous MDD is denoted by* **MDD(k)**, *where $k = k_1 = k_2 = \ldots = k_u$. An MDD($k$) represents a mapping $f : P^u \rightarrow B$, while a heterogeneous MDD represents a mapping $f : P_1 \times P_2 \times \ldots \times P_u \rightarrow B$, where $P = \{0, 1, \ldots, 2^k - 1\}$, $P_i = \{0, 1, \ldots, 2^{k_i} - 1\}$, and $B = \{0, 1\}$.*



(a) BDD (b) MDD(2)

Fig. 1. BDD and MDD(2)



(a) $X_1 = (x_1, x_2, x_3)$, $X_2 = (x4)$ (b) $X_1 = (x_1)$, $X_2 = (x_2, x_3, x_4)$

Fig. 2. Heterogeneous MDDs

In an MDD(k), non-terminal nodes have 2^k edges. When $k = 1$, an MDD(1) is an ROBDD. In a heterogeneous MDD, non-terminal nodes representing a super variable X_i have 2^{k_i} edges, where k_i is the number of elements in X_i.

Definition 2 *In a decision diagram (DD), the* **number of nodes in the DD***, denoted by $nodes(DD)$, includes only non-terminal nodes.*

Definition 3 *The* **width of a DD with respect to the super variable** X_i*, denoted by $width(DD, i)$, is the number of nodes in the DD corresponding to X_i.*

The number of nodes in the MDD with the partition $(X_1, X_2, \ldots, X_u)$ is given by

$$nodes(MDD) = \sum_{i=1}^{u} width(MDD, i).$$

Example 2 *Consider the function:*

$$f = x_1x_2x_3 \vee x_2x_3x_4 \vee x_3x_4x_1 \vee x_4x_1x_2.$$

Fig. 1(a), Fig. 1(b), and Fig. 2 represent the ROBDD, the MDD(2), and the heterogeneous MDDs for the function, respectively. In Fig. 1(a), the solid lines and the dotted lines denote 1-edges and 0-edges, respectively. In Fig. 1(b), the input variables $X = (x_1, x_2, x_3, x_4)$ are partitioned into (X_1, X_2), where $X_1 = (x_1, x_2)$ and $X_2 = (x_3, x_4)$. In Fig. 2(a), $X_1 = (x_1, x_2, x_3), X_2 = (x_4)$. And, in Fig. 2(b), $X_1 = (x_1), X_2 = (x_2, x_3, x_4)$.

(End of Example)

C Representations of Multiple-Output Functions

Logic networks usually have many outputs. In most cases, independent representation of each output is inefficient. Let the multiple-output functions be $F = (f_0, f_1, \ldots, f_{m-1})$: $B^n \rightarrow B^m$, where $B = \{1, 0\}$, and n and m denote the number of input and output variables, respectively. Several methods exist to represent multiple-output functions using BDDs [15, 20, 21, 22]. In this paper, we use Shared Binary Decision Diagrams (SBDDs) [20] to represent multiple-output functions. In the following, a BDD means a Shared ROBDD (SROBDD) unless stated otherwise.

III Average Path Length (APL)

Definition 4 *A* **path in a DD** *is a sequence of nodes for some assignment of values to all variables. The* **path length** *is the number of non-terminal nodes on the path.*

Definition 5 *In a DD, the* **node traversing probability***, denoted by $prob(DD, i)$, is a fraction of all assignments of values to variables whose path includes v_i.*

In an MDD, we assume the following computation model:

1. DDs for logic functions are evaluated by traversing nodes from the root node to a terminal node according to values of the input variables.
2. MDDs are implemented directly, not simulated using the BDD package as described in [11].
3. Encoded input values are available, and their access time is negligible. For example, when $X_1 = (x_1, x_2, x_3, x_4) = (1, 0, 0, 1)$, $X_1 = 9$ is available as an input to the algorithm.
4. Most computation time is spent for accessing nodes.
5. The access time to all MDD nodes is equal.

In this case, the time to evaluate a DD for a logic function is proportional to the number of non-terminal nodes on the path (i.e., path length). And also, we assume that each binary variable occurs as a 0 with the same probability as a 1. Under these assumptions, we use the **average path length** (APL) to estimate the evaluation time of different types of DDs.

In this paper, we use a Shared Decision Diagram (SDD) to represent multiple-output functions $F = (f_0, f_1, \ldots, f_{m-1})$. The APL of an SDD is the sum of the APLs of individual DDs for each function f_i [23].

Theorem 1 *[23] The APL of a DD is given by the sum of the node traversing probabilities of all the non-terminal nodes.*

Theorem 2 *For a BDD and a heterogeneous MDD that represent the same logic functions, the following relation holds:*

$$(\textit{APL of a heterogeneous MDD}) \leq (\textit{APL of a BDD}).$$

(Proof) The number of non-terminal nodes on the path never increases by grouping of variables in X. Therefore, we have the theorem. (Q.E.D)

IV CODE GENERATION

Two methods exist to generate the code from heterogeneous MDD.

A Compiled Code Method

In the compiled code method, a **branching program** [1] is generated directly from a DD by replacing each node with **if** and **goto** statements, or **switch** and **goto** statements. For BDDs, this method produces an efficient code. However, for MDDs, as shown in the following example, this method may produce slow code because some C-compilers generate inefficient code.

Example 3 *Fig. 3(a) shows a node in an MDD, and Fig. 3(b) shows the branching program implemented with a switch statement. Some compilers generate code whose execution time depends on the value of X_i. When $X_i = 0$, only one comparison of X_i with 0 is needed. When $X_i = k-1$, $k-1$ comparisons of X_i with the values in each "case" statement are needed. (End of Example)*

Also, in this method, different logic function results in different code.

B Data Table Method

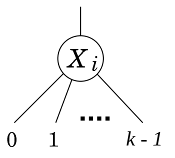
In the data table method, from a heterogeneous MDD, we generate a **data table** such as Fig. 3(c). Also, we use the **traversing program** that evaluates the data table as follows:

1. Read an index i in the data table.
2. Obtain the value of X_i.
3. (Address in the data table) $\leftarrow$ ((Address in the data table) + (value of X_i) + 1).
4. Read the data from the given address in the table.

In this method, the traversing time of all the MDD nodes is the same, and the computation time of the codes depends only on the APL of an MDD. For a different function, we need different data tables, while the traversing programs are the same.

The memory size needed for this method is

$$\text{(the code size of the traversing program)} + \text{(the memory size needed for the data table)}.$$



(a) A node in an MDD

```
switch($X_i$){
   case 0 :
      goto 0-edge pointer;
   case 1 :
      goto 1-edge pointer;
   ...
   case $k$-1 :
      goto $k$-1-edge pointer;
}
```

(b) Compiled code method

index i of X_i
0-edge pointer
1-edge pointer
2-edge pointer
...
k-1-edge pointer

(c) Data table method

Fig. 3. Implementation of a node in an MDD

The memory size for the data table is

$$\text{(the bit width of a word)} \times \left[\sum_{i=1}^{u} \left((2^{k_i}+1) \times width(\text{heterogeneous MDD}, i)\right)\right],$$

where the sum in square brackets denotes the number of words to represent a heterogeneous MDD.

C Optimization of Codes

As mentioned in the previous section, the evaluation time of the code generated from heterogeneous MDDs depends on the APL of heterogeneous MDDs. Therefore, the minimization of the APL also minimizes the computation time of the code.

When the order of the input variables $X = (x_1, x_2, \ldots, x_n)$ is fixed, the APL for a heterogeneous MDD is determined by the partition of the input variables X.

Example 4 *The APLs for different DDs are as follows: For the BDD in Fig. 1(a), 3.125; for the MDD(2) in Fig. 1(b), 1.75; for the heterogeneous MDD in Fig. 2(a), 1.375; and for the heterogeneous MDD in Fig. 2(b), 2.0. (End of Example)*

The partition of X that minimizes the APL is the trivial partition, $X = X_1$, where $k_1 = n$. However, the memory size needed to represent the heterogeneous MDD for the trivial partition is $(2^n + 1) \times width(\text{heterogeneous MDD}, 1)$, and is too large in most cases. Therefore, we find a partition of X that minimizes the APL within the given memory size. We formulate the minimization problem for the APL as follows:

Problem 1 *Suppose that the variable order for the BDD is fixed. Given a BDD for the logic function f and its memory size L, find a partition of X that makes the APL of the heterogeneous MDD minimum within the memory size L.*

In Algorithm 1, we show a pseudo-code to solve Problem 1. This algorithm is based on a branch-and-bound strategy using a caching technique to reduce the computation time. The sub-solutions are stored in the cache, but only a subset of sub-solutions is kept in it because the number of sub-solutions is too large in many cases. In other words, this algorithm is similar to the dynamic programing, except for that the cache is overwritten. Note that the memory size is measured by the total number of words to represent the heterogeneous MDD. For simplicity, we assume that the variable order is $x_1, x_2, \ldots, x_n$. This algorithm uses the index of the top variable for the BDD and the memory size L as the arguments.

Algorithm 1 *(Minimization of APL)*

```
1: minimize_APL (index $i$, mem_size $l$) {
2:    if ($i > n$)
3:       return 0 ;
4:    **check the cache** ;
5:    if (cache.index == $i$ && cache.mem == $l$) {
6:       **register the partition cache.k** ;
7:       return cache.APL ;
8:    }
9:    min_APL = (APL for BDD) ;
10:   for (k = $n - i + 1$; k $\geq$ 1; k$--$) {
11:      memory = $(2^k + 1) \times width($heterogeneous MDD$, j)$ ;
12:      next_index $i'$ = $i$ + k ;
13:      if (($l$ $-$ memory) $<$ **lower_bound[$i'$]**)
14:         continue ;
15:      current_APL = 0 ;
16:      for (all nodes $v$ representing $X_j$)
17:         current_APL += $prob($heterogeneous MDD$, v)$ ;
18:      current_APL += minimize_APL ($i'$, $l$ $-$ memory) ;
19:      if (current_APL $<$ min_APL) {
20:         **register the partition k** ;
21:         min_APL = current_APL ;
22:      }
23:   }
24:   **store (overwrite) to the cache** ;
25:   return min_APL ;
26:}
```

This algorithm produces an optimum solution by calculating the APLs for different partitions of X. The calculation of the APL uses Theorem 1. To compute the node traversing probability $prob($heterogeneous MDD$, v)$ of the 17th line, we used the method in [23]. The 13th line uses **lower bounds** on the memory size [18] to reduce computation time.

V Experiment and Observation

Experiments were done using the following environment:

- CPU: Pentium4 Xeon 2.8GHz,
- L1 Cache: 32KB, L2 Cache: 512KB, Memory: 4GB,
- OS: Redhat (Linux 7.3),
- C-compiler: gcc -O2.

A Computation Time to Optimize Code

Table I shows the computation time to optimize code. Algorithm 1 obtained heterogeneous MDDs from BDDs with the memory size L needed to represent the BDDs, where the size of the cache to store the sub-solutions is 500,000. The column "I/O" denotes the numbers of the inputs and outputs of the benchmark circuits; the column "BDD" denotes the results for BDDs (i.e. without optimizing code); and, the column "MDD" denotes the results for heterogeneous MDDs. The column "Size" denotes the number of words to represent the DD, the column "APL" denotes the APL for the DD, and the column "Time" denotes the computation time, in seconds, to obtain an optimum heterogeneous MDD. In Table I, the bottom row "Average of ratios" denotes the arithmetic average of ratios of the results for each DD, where the results for BDD are set to 1.00.

Table I shows that the code can be optimized in short computation time. Algorithm 1 could obtain the optimum solutions for logic functions with up to 1,500 inputs by using larger cache in reasonable time. However, the experimental results for those functions were omitted from Table I because such functions are rarely used in the embedded system.

B Evaluation Time and Code Size

To show the performance of code generated by heterogeneous MDDs, we compared the code generated by the data table method with the code generated by the following two methods.

LCC: Code generated by Levelized Compiled Code (LCC) method. The code is composed of fragments of the code translated from modules in the multi-level networks. And, the fragments of code are executed in the topological order from the inputs toward the outputs. LCC method is often used in **cycle-based simulation**. We used SIS [24] with the script "script.algebraic" to simplify multi-level logic networks.

BDD: The code generated from BDDs by the data table method. We used the "sifting algorithm [19]" to reduce the number of nodes in BDDs.

Table II compares evaluation time and code sizes for the code generated by three different methods. The column "Evaluation time" in Table II denotes the average evaluation time, in nanoseconds, for a single random input vector. The column "Code size" denotes the object code size, in bytes, of programs excluding code that reads the input vectors and displays the outputs. The code sizes of BDDs and heterogeneous MDDs were calculated by: *(the code size of the traversing program) + (the memory size of the data table)*, where the bit width of the data table is 4 Bytes. The columns "LCC", "BDD", and "MDD" denote results for the code generated by LCC method, BDDs, and heterogeneous MDDs, respectively. The column "Literals" denotes the numbers of literals in multi-level logic networks; the column "Nodes" denotes the numbers of non-terminal nodes in BDDs; and, the column "**LN-ratio**" denotes

TABLE I
SIZE AND APL FOR BDDS AND HETEROGENEOUS MDDS

		BDD		MDD		
Name	I/O	Size	APL	Size	APL	Time
5xp1	7 / 10	204	32.03	199	17.84	0.01
9sym	9 / 1	99	7.34	93	2.22	0.01
C499	41/ 32	83529	782.66	83498	201.72	0.38
adr8	16/ 9	195	43.02	191	18.05	0.01
adr9	18/ 10	222	49.01	206	21.02	0.01
alu4	14/ 8	1380	40.95	1369	15.50	0.01
apex1	45/ 45	3834	181.81	3824	130.99	0.01
apex2	39/ 3	177	10.90	177	6.23	0.01
apex3	54/ 50	2796	188.58	2794	92.31	0.01
apex4	9 / 19	2910	113.34	2392	27.27	0.01
b12	15/ 9	168	30.30	168	24.13	0.01
bw	5 / 28	324	96.56	287	49.13	0.01
clip	9 / 5	315	28.30	311	11.73	0.01
comp	32/ 3	420	12.00	414	3.56	0.01
con1	7 / 2	45	6.69	42	4.44	0.01
cordic	23/ 2	225	17.21	218	5.66	0.01
cps	24/109	2970	291.89	2964	165.32	0.01
duke2	22/ 29	1095	87.89	1091	53.47	0.01
ex1010	10/ 10	4236	86.71	2640	17.21	0.01
ex5	8 / 63	834	173.05	833	102.48	0.01
inc	7 / 9	210	28.53	201	12.94	0.01
misex1	8 / 7	108	24.59	108	24.34	0.01
misex2	25/ 18	243	41.62	243	38.94	0.01
misex3	14/ 14	1626	83.90	1621	28.48	0.01
my_adder	33/ 17	429	94.00	425	36.51	0.01
pdc	16/ 40	1665	137.37	1655	77.57	0.01
rd53	5 / 3	69	13.00	57	4.75	0.01
rd73	7 / 3	129	19.31	129	5.63	0.01
rd84	8 / 4	177	24.15	173	8.85	0.01
sao2	10/ 4	255	15.35	253	7.46	0.01
seq	41/ 35	3744	100.02	3723	51.45	0.01
spla	16/ 46	1743	136.58	1713	83.86	0.01
squar5	5 / 8	111	21.56	105	14.56	0.01
t481	16/ 1	96	9.00	92	3.31	0.01
table3	14/ 14	2253	90.46	2128	23.30	0.01
table5	17/ 15	2007	86.09	1996	25.79	0.01
vg2	25/ 8	243	34.88	242	20.01	0.01
xor5	5 / 1	27	5.00	23	2.00	0.01
Average of ratios		1.00	1.00	0.96	0.48	–

(the number of literals) / (the number of nodes). In Table II, the bottom row “Average of ratios” denotes the arithmetic average of ratios for each method, where the average for LCC method is set to 1.00. Note that these averages were calculated from the functions, except for $C499$ and $C6288$. We could not construct the BDD and the heterogeneous MDD for $C6288$ due to the memory size overflow. For many benchmark functions, heterogeneous MDDs generated the fastest and the most compact code. However, for some functions, the LCC method generated faster and more compact code than heterogeneous MDDs.

C Observation

For some functions, heterogeneous MDDs produce efficient code, while for others, LCC methods produce efficient codes. To find the logic functions that are suitable for the heterogeneous MDD code generation method, we introduced the **LN-ratio**, the ratio of the number of literals to the number of nodes. Table II allows us to make the following:

Observation 1 *When the LN-ratio is greater than two, heterogeneous MDDs generate smaller code, while when the LN-ratio is less than one, the LCC method generates a smaller code.*

Therefore, the LN-ratio can be used as a rule of thumb to find a suitable code generation method.

VI CONCLUSION AND COMMENTS

In this paper, we proposed a method to generate code for embedded systems using heterogeneous MDDs. We presented

TABLE II
COMPARISON OF EVALUATION TIME AND CODE SIZE

				LN-	Evaluation time			Code size		
Name	I/O	Literals	Nodes	ratio	LCC	BDD	MDD	LCC	BDD	MDD
5xp1	7 / 10	133	68	1.96	250	179	92	2668	1922	1894
9sym	9 / 1	254	33	7.70	597	36	22	4196	1488	1452
C499	41/ 32	558	27843	0.02	1793	8491	4757	7000	335398	335218
C6288	32/ 32	3630	–	–	20399	–	–	48848	–	–
adr8	16/ 9	106	65	1.63	209	225	113	2244	1902	1866
adr9	18/ 10	120	74	1.62	247	250	136	2420	2016	1930
alu4	14/ 8	387	460	0.84	1029	240	94	6012	6636	6574
apex1	45/ 45	1272	1278	1.00	3715	1426	1126	18280	16588	16514
apex2	39/ 3	3874	59	65.66	12172	81	57	53376	1846	1822
apex3	54/ 50	1633	932	1.75	5183	1458	860	22828	12464	12432
apex4	9 / 19	2732	970	2.82	11895	716	207	34940	12768	10682
b12	15/ 9	99	56	1.77	202	207	171	2500	1792	1784
bw	5 / 28	212	108	1.96	335	613	276	3936	2434	2280
clip	9 / 5	529	105	5.04	1298	161	69	8024	2360	2334
comp	32/ 3	159	140	1.14	433	55	33	3016	2822	2760
con1	7 / 2	23	15	1.53	64	45	28	1216	1270	1254
cordic	23/ 2	2998	75	39.97	6987	86	28	40816	2022	1962
cps	24/109	1062	990	1.07	2887	2614	1787	16548	13218	13172
duke2	22/ 29	456	365	1.25	939	650	449	7032	5554	5524
ex1010	10/ 10	1624	1412	1.15	5521	575	129	21844	18056	11656
ex5	8 / 63	592	278	2.13	1472	1398	924	8932	4550	4540
inc	7 / 9	141	70	2.01	225	166	93	2876	1944	1900
misex1	8 / 7	67	36	1.86	126	130	134	1868	1534	1532
misex2	25/ 18	113	81	1.40	255	325	296	2932	2130	2118
misex3	14/ 14	3020	542	5.57	10051	483	237	41304	7632	7594
my_adder	33/17	224	143	1.57	632	467	247	3692	2888	2828
pdc	16/ 40	393	555	0.71	969	1025	727	6752	7844	7792
rd53	5 / 3	65	23	2.83	96	56	32	1820	1364	1310
rd73	7 / 3	142	43	3.30	280	75	29	2636	1608	1598
rd84	8 / 4	181	59	3.07	345	112	46	3104	1804	1778
sao2	10/ 4	185	85	2.18	414	90	47	3472	2120	2102
seq	41/ 35	1652	1248	1.32	5008	687	448	23624	16200	16080
spla	16/ 46	1209	581	2.08	2992	1072	708	17104	8168	8032
squar5	5 / 8	76	37	2.05	119	120	85	2200	1542	1514
t481	16/ 1	46	32	1.44	97	48	25	1368	1490	1452
table3	14/ 14	970	751	1.29	2324	542	221	13352	10140	9620
table5	17/ 15	925	669	1.38	2330	536	239	12972	9164	9100
vg2	25/ 8	97	81	1.20	213	229	135	2516	2110	2082
xor5	5 / 1	16	9	1.78	36	25	19	1008	1192	1170
Average of ratios					1.00	0.56	0.36	1.00	0.69	0.67

a minimization algorithm for the APL, and a method to generate efficient code from heterogeneous MDDs. Also, we introduced the LN-ratio, which shows a suitable code generation method. Our experimental results with many benchmark functions show that: 1) This method generates more efficient code than BDDs; 2) This method generates efficient code for the functions whose LN-ratios are greater than two; 3) LCC method generates efficient code for the functions whose LN-ratios are less than one.

Note that the heterogeneous MDD generates smaller and faster code relatively easily in a short computation time. The code to evaluate data table can be easily implemented in dedicated hardware for faster evaluation.

Algorithm 1 can obtain optimum solutions for logic functions with up to 1,500 inputs in reasonable time, assuming the fixed initial BDD variable order. However, to obtain the optimum heterogeneous MDDs that consider both the partition of the input variables and the ordering of the variables, we need to improve the algorithms and heuristics.

ACKNOWLEDGMENTS

This research is partly supported by the Grant in Aid for Scientific Research of The Japan Society for the Promotion of Science (JSPS), and the Takeda Foundation. Discussion with Dr. Alan Mishchenko and Prof. Jon. T. Butler improved the presentation of the paper.

REFERENCES

[1] P. Ashar and S. Malik, "Fast functional simulation using branching programs," *ICCAD'95*, pp. 408–412, Nov. 1995.

[2] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki, “Synthesis of software programs for embedded control applications,” *IEEE Trans. CAD*, Vol. 18, No. 6, pp.834-849, June 1999.

[3] B. Becker and R. Drechsler, “Efficient graph based representation of multivalued functions with an application to genetic algorithms,” *Proc. of International Symposium on Multiple Valued Logic*, pp. 40-45, May 1994.

[4] R. P. Brent and H. T. Kung, “The area-time complexity of binary multiplication,” *Journal of the ACM*, Vol. 28, No. 3, pp. 521-534, July 1981.

[5] F. Brglez and H. Fujiwara, “Neutral netlist of ten combinational benchmark circuits and a target translator in FORTRAN,” *Special session on ATPG and fault simulation, Proc. IEEE Int. Symp. Circuits and Systems*, June 1985, pp. 663-698.

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

[7] W. Günther and R. Drechsler, “Minimization of free BDDs,“ *Asia and South Pacific Design Automation Conference (ASP-DAC’99)*, pp.323-326, Jan. 18-21, 1999, Wanchai, Hong Kong.

[8] W. Günther, “Minimization of free BDDs using evolutionary techniques,“ *International Workshop on Logic Synthesis 2000 (IWLS-2000)*, pp.167-172, May 31-June 2, 2000, Loguna Cliffs Marriott, Dana Point, CA.

[9] Y. Iguchi, T. Sasao, M. Matsuura, and A. Iseno “A hardware simulation engine based on decision diagrams,” *Asia and South Pacific Design Automation Conference (ASP-DAC’2000)*, Jan. 26-28, Yokohama, Japan.

[10] Y. Iguchi, T. Sasao, M. Matsuura, “Implementation of multiple-output functions using PQMDDs,” *International Symposium on Multiple-Valued Logic*, pp.199-205, May 2000.

[11] T. Kam, T. Villa, R. K. Brayton, and A. L. Sagiovanni-Vincentelli, “Multi-valued decision diagrams: Theory and Applications,” *Multiple-Valued Logic*, 1988, Vol. 4, No. 1-2, pp. 9–62, 1998.

[12] C. Kim, L. Lavagno, and A. S-Vincentelli, “Free BDD-based software optimization techniques for embedded systems,” *Design, Automation and Test in Europe (DATE2000)*, Paris, pp.14-19, March 2000.

[13] H.-T. Liaw, and C.-S. Lin. “On the OBDD-representation of general Boolean function,” *IEEE Transactions on Computers*, Vol. 4, No. 6, pp. 661–664, June 1992.

[14] P. C. McGeer, K. L. McMillan, A. Saldanha, A. L. Sangiovanni-Vincentelli, and P. Scaglia, “Fast discrete function evaluation using decision diagrams,” *ICCAD’95*, pp. 402–407, Nov. 1995.

[15] S. Minato, N. Ishiura, and S. Yajima, “Shared binary decision diagram with attributed edges for efficient Boolean function manipulation,” *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 52–57, June 1990.

[16] D. M Miller, “Multiple-valued logic design tools,” *Proc. of International Symposium on Multiple Valued Logic*, pp. 2–11, May 1993.

[17] S. Nagayama, T. Sasao, Y. Iguchi, and M. Matsuura, “Representations of logic functions using QRMDDs,” *Proc. of International Symposium on Multiple-Valued Logic*, pp. 261-267, Boston, Massachusetts, U.S.A, May 15-18, 2002.

[18] S. Nagayama, and T. Sasao, “Compact representations of logic functions using heterogeneous MDDs,” *Proc. of International Symposium on Multiple-Valued Logic*, Tokyo, Japan, May, 2003 (accepted for publication).

[19] R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” ICCAD ’93, pp. 42–47.

[20] T. Sasao and M. Fujita (ed.), *Representations of Discrete Functions*, Kluwer Academic Publishers 1996.

[21] T. Sasao and J. T. Butler, “A method to represent multiple-output switching functions by using multivalued decision diagrams,” *Proc. of International Symposium on Multiple-Valued Logic*, pp. 248-254, Santiago de Compostela, Spain, May 29-31, 1996.

[22] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.

[23] T. Sasao, Y.Iguchi, and M. Matsuura, “Comparison of decision diagrams for multiple-output logic functions,” *International Workshop on Logic and Synthesis*, New Orleans, Louisiana, June 4-7, 2002, pp.379-384.

[24] E. M. Sentovich et. al., “SIS: A System for Sequential Circuit Synthesis,“ Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, 1992, http://www-cad.eecs.berkeley.edu/Software/software.html.

[25] F. Somenzi, “CUDD: CU Decision Diagram Package Release 2.3.1,“ University of Colorado at Boulder, 2001.

[26] S. Yang, *Logic synthesis and optimization benchmark user guide version 3.0*, MCNC, Jan. 1991.

[27] Y. Jiang and B. K. Brayton, “Software synthesis from synchronous specifications using logic simulation techniques,” *Design Automation Conference*, pp. 319-324, New Orleans, LA, U.S.A, June 10-14, 2002.