PAPER Special Section on VLSI Design and CAD Algorithms

Bi-Partition of Shared Binary Decision Diagrams

Munehiro MATSUURA†, *Nonmember*, **Tsutomu SASAO**††a), *Regular Member*, **Jon T. BUTLER**†††, *Nonmember*, *and* **Yukihiro IGUCHI**††††, *Regular Member*

SUMMARY A shared binary decision diagram (SBDD) represents a multiple-output function, where nodes are shared among BDDs representing the various outputs. A partitioned SBDD consists of two or more SBDDs that share nodes. The separate SBDDs are optimized independently, often resulting in a reduction in the number of nodes over a single SBDD. We show a method for partitioning a single SBDD into two parts that reduces the node count. Among the benchmark functions tested, a node reduction of up to 23% is realized.

key words: *shared binary decision diagram, SBDD, bi-partition, multiple-output function, decomposition*

1. Introduction

Various methods exist to represent multiple-output functions [15]–[18]. Among them, shared binary decision diagrams (SBDDs) [4], [11] are most commonly used, since their sizes are usually smaller [18] than other types of BDDs, such as multi-terminal binary decision diagrams (MTBDDs) [16] and BDDs for characteristic functions (BDDs for CFs) [1], [19]. Some authors [5] use the term "multi-rooted BDD" instead of SBDD. However, for some applications, SBDDs are still too large and more compact representations are required.

To further reduce memory storage we propose partitioned SBDDs, as a method to represent multiple-output functions. Each part represents a set of outputs, and is optimized independently. Such BDDs are considered as a special case of partitioned BDDs [6], [12], [13] and free BDDs (FBDDs) [7], [8]. Note that BDD nomenclature is not unified. For example, the term "partitioned BDDs" has a different meaning for certain authors [20].

Manuscript received March 13, 2002.
Manuscript revised June 24, 2002.
Final manuscript received August 14, 2002.

†The author is with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka-shi, 820-8502 Japan.
††The author is with the Department of Computer Science and Electronics, and Center for Microelectronic Systems, Kyushu Institute of Technology, Iizuka-shi, 820-8502 Japan.
†††The author is with the Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA 93943, U.S.A.
††††The author is with the Department of Computer Science, Meiji University, Kawasaki-shi, 214-8571 Japan.
a) E-mail: sasao@cse.kyutech.ac.jp

Applications of partitioned SBDDs are similar to that of partitioned BDDs and FBDDs. When applied to hardware synthesis, one replaces each non-terminal node of an SBDD by a multiplexer (MUX), forming a network for F. This is used to design multiplexer-type FPGAs [3] and pass-transistor logic [22]. In such applications, minimizing the number of nodes in the SBDD also minimizes the hardware required in the implementation.

Although our goal is a large reduction in the node count, our results offer a design alternative when the reduction is small. Since a bi-partition yields two separate SBDDs, these can be implemented independently, allowing a more flexible layout. The advantage of this may be even more significant than the node count reduction [9].

2. Partition of SBDDs

In this paper, we consider ordered SBDDs, where the input variables appear in the same order along all paths through the graph beginning from a root node and ending on a leaf node. Further, in this part of the paper, we assume that, in all such paths, no variable appears more than once.

An SBDD is considered as a compact BDD representation of a multiple-output function, since nodes can be shared among many outputs [11].

Example 2.1: Consider the two-output function:

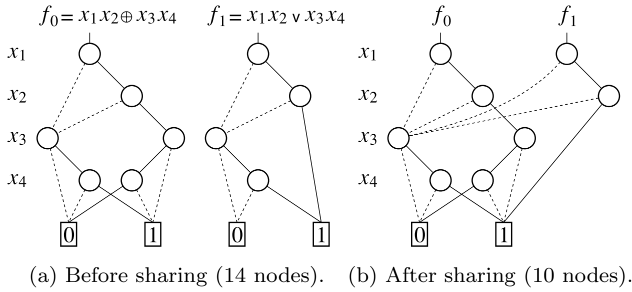
$$f_0 = x_1x_2 \oplus x_3x_4,$$
$$f_1 = x_1x_2 \vee x_3x_4.$$

In this case, $\pi = (x_1, x_2, x_3, x_4)$ is a good ordering of the input variables for both f_0 and f_1. Note that some nodes can be shared between f_0 and f_1, as shown in Fig. 1. In the figures, dotted lines denote 0-edges, while solid lines denote 1-edges. (End of Example)

In an SBDD, there can be only one ordering of input variables for all output functions. Thus, the size tends to be large when the individual functions have *different* optimal orderings of the input variables.

Example 2.2: Consider the BDDs of functions:

$$f_0 = x_1x_2 \vee x_3x_4 \vee x_5x_6,$$



(a) Before sharing (14 nodes). (b) After sharing (10 nodes).

Fig. 1 Shared BDD.

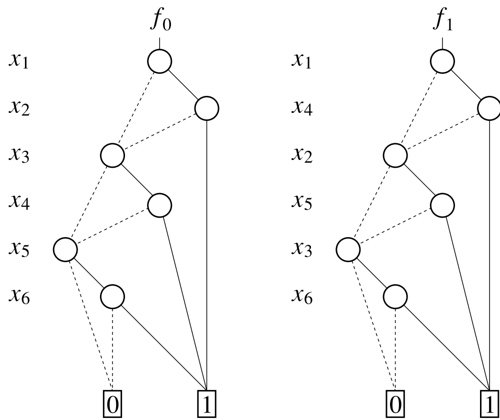


Fig. 2 A pair of BDDs that has fewer nodes than an optimized monolithic SBDD.

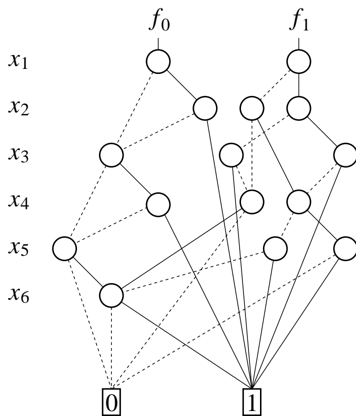


Fig. 3 An optimized monolithic SBDD.

$$f_1 = x_1x_4 \vee x_2x_5 \vee x_3x_6.$$

In this case, $\pi_0 = (x_1, x_2, x_3, x_4, x_5, x_6)$ is an optimal ordering for f_0, while $\pi_1 = (x_1, x_4, x_2, x_5, x_3, x_6)$ is an optimal ordering for f_1. Figure 2 shows the corresponding BDDs. Together, they require a total of $8 \times 2 = 16$ nodes. On the other hand, a minimum SBDD for $\{f_0, f_1\}$ requires 17 nodes, as shown in Fig. 3. In this case, the pair of separately optimized BDDs is smaller than the optimized monolithic SBDD for $\{f_0, f_1\}$. This is an example of a partitioned BDD that is smaller than the monolithic SBDD. (End of Example)

From these examples, we can formulate

Problem 2.1: (Partitioned SBDD)
Given a multiple-output function F, represent F by a set of SBDDs so that the total number of nodes is minimized, where each SBDD is optimized independently.

3. Bi-Partition of SBDDs

In this section, we show an exact and a heuristic method for solving the partitioned SBDD problem when there are no more than two parts.

Definition 3.1: Let $F = \{f_0, f_1, \ldots, f_{m-1}\}$ be the set of the output functions. $size(SBDD, F, \pi)$ denotes the number of nodes in the SBDD for F, where π is the ordering of the input variables. $size(SBDD, F)$ denotes the minimum $size(SBDD, F, \pi)$ for F over all orderings π.

Then, we can formulate

Problem 3.2: (Bi-partitioned SBDD)
Given a multiple-output function $F = \{f_0, f_1, \ldots, f_{m-1}\}$, represent F by a pair of SBDDs so that $size(SBDD, F_1) + size(SBDD, F_2)$ is minimized, where $F_1 \cup F_2 = F$, $F_1 \cap F_2 = \phi$, and $F_1 \neq \phi$, where ϕ is the null set.

It is possible that, for all non-trivial bi-partitions, the total number of nodes in the partitioned SBDD is greater than in the original one. In this case, we accept the original given SBDD as the best we can do. This is represented as the trivial partition $F = (F, \phi)$. For example, we choose $F = (F, \phi)$ to realize the functions in Example 2.1 since the only non-trivial partition produces a larger node count.

Algorithm 3.1: (Bi-partition of an SBDD: Exact method)

1. $min_size \leftarrow \infty$.
2. Enumerate a bi-partition $\{F_1, F_2\}$ of $F = \{f_0, f_1, \ldots, f_{m-1}\}$ (where $F_1 \cup F_2 = F$ and $F_1 \cap F_2 = \phi$). If there are no more bi-partitions, stop.
3. $size \leftarrow size(SBDD, F_1) + size(SBDD, F_2)$.
4. If $(size < min_size)$, then $min_size \leftarrow size$.
5. Go to 2.

Although Algorithm 3.1 produces an exact minimum solution, it requires $T = 2^{m-1}$ minimizations of pairs of SBDDs, since T is the number of bi-partitions on F. So, this method is only practical for functions with small n and m. The following is a heuristic algorithm that can be used for functions with large BDDs.

Algorithm 3.2: (Bi-partition of an SBDD: Heuristic method)

1. Simplify the SBDD for F by using a heuristic method, i.e. [14]. Let π be an ordering of the input variables that simplifies the SBDD for F.

2. Simplify the BDD for each component function f_i $(i = 0, 1, \ldots, m-1)$ using the method of [14]. Let $r_i = \dfrac{size(BDD, f_i)}{size(BDD, f_i, \pi)}$, where $size(BDD, f_i)$ is the number of nodes in the minimum BDD for f_i and $size(BDD, f_i, \pi)$ is the number of nodes in the BDD for f_i under the ordering π.
3. $r_{av} = \dfrac{1}{m}\sum_{i=0}^{m-1} r_i$, $F_1 \leftarrow \phi$, and $F_2 \leftarrow \phi$.
   For each f_i
   if $(r_i \leq r_{av})$ then
   $F_1 \leftarrow F_1 \cup \{f_i\}$
   else
   $F_2 \leftarrow F_2 \cup \{f_i\}$.
4. Minimize the SBDDs of F_1 and F_2. $e_1 \leftarrow size(SBDD, F_1) + size(SBDD, F_2)$ using the method of [14].
5. Let f_h be a function that has the minimal r_i in F_2.
6. Minimize the SBDDs of $F_1 \cup \{f_h\}$ and $F_2 - \{f_h\}$. $e_2 \leftarrow size(SBDD, F_1 \cup \{f_h\}) + size(SBDD, F_2 - \{f_h\})$ using the method of [14].
   If $(e_1 > e_2)$ then
   $e_1 \leftarrow e_2$
   $F_1 \leftarrow F_1 \cup \{f_h\}$
   $F_2 \leftarrow F_2 - \{f_h\}$
   go to 5
7. Let f_h be a function that has the maximal r_i in F_1.
8. Minimize the SBDDs of $F_2 \cup \{f_h\}$ and $F_1 - \{f_h\}$. $e_2 \leftarrow size(SBDD, F_2 \cup \{f_h\}) + size(SBDD, F_1 - \{f_h\})$ using the method of [14].
   If $(e_1 > e_2)$ then
   $e_1 \leftarrow e_2$
   $F_1 \leftarrow F_1 - \{f_h\}$
   $F_2 \leftarrow F_2 \cup \{f_h\}$
   go to 7
9. Stop.

The idea of the algorithm is as follows:

1) Let π_i be an optimal ordering of the input variables for a BDD that represents function f_i. Let π be an optimal ordering of the input variables for an SBDD that represents all functions in F.
2) When the ratio $\dfrac{size(BDD, f_i)}{size(BDD, f_i, \pi)}$ is large the ordering π is near optimal for f_i. On the other hand, when it is small, π is far from optimal.
3) In Step 3, we partition F into two sets F_1 and F_2. F_1 consists of the functions for which π is a good ordering, and F_2 consists of functions for which π is not a good ordering.
4) In Steps 5–8, we improve the partition by moving functions likely to produce an improvement from one set to the other.

4. Node Sharing

In this section, we consider the case where nodes are shared across SBDDs.

Example 4.1: In Fig. 2, the node labeled x_6 and the constant nodes from the two BDDs can be combined yielding a BDD with 13 nodes and reducing the node count by 3. However, the resulting BDD is not an SBDD because of different orderings for input variables across middle level nodes. (End of Example)

As shown in the above example, *node sharing* can produce a BDD that is not an SBDD. Thus, we cannot use existing BDD packages. Therefore, we perform this operation as a separate process.

Proposition 4.1: Let v_0 and v_1 be nodes in two SBDDs: SBDD0 and SBDD1, respectively. If v_0 and v_1 represent the same logic function, then one can be removed.

This is a sufficient condition to share a node between two SBDDs. The following example shows a case where two nodes representing *different* functions can be shared.

Example 4.2: Consider the two functions:

$$f_0 = (\bar{x}_1 x_2 \vee x_1 \bar{x}_2) x_3,$$
$$f_1 = x_1 \bar{x}_2 x_3 \vee \bar{x}_1 x_2 \bar{x}_3.$$

Figure 4 shows a pair of BDDs representing f_0 and f_1. Note that v_0 represents f_0, and v_1 represents the function $x_1 x_3$. Figure 5 shows the BDD after node sharing. Note that v_0 is used instead of v_1 in the BDD

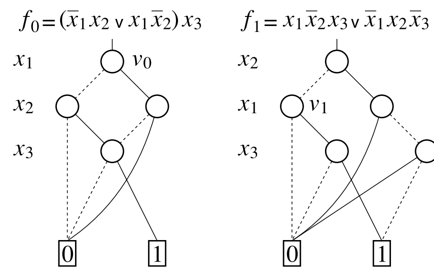


Fig. 4 Pair of BDDs.

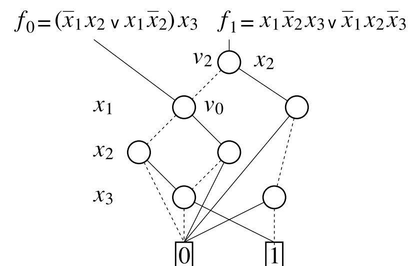


Fig. 5 Sharing nodes that represent different functions.

of f_1. Indeed, v_2 represents the function f_1 since,

$$\bar{x}_2 f_0 \vee x_2\bar{x}_1\bar{x}_3 = x_1\bar{x}_2x_3 \vee x_2\bar{x}_1\bar{x}_3 = f_1.$$

Note that a significant savings has occurred. The combined BDD has 9 nodes, while the separate BDDs have a total of 13 nodes. Note also that the BDD in Fig. 5 is not an SBDD, since x_2 occurs twice in paths from the node for v_2 to the constant nodes. (End of Example)

This example shows a way to reduce significantly the number of nodes in bi-partitioned SBDDs. It is, however, difficult to apply since it depends on subtle algebraic equivalences. In the interest of an efficient algorithm, we ignore the above case, and we only use Proposition 4.1 for node sharing between two SBDDs.

Algorithm 4.1: (Node sharing between two SBDDs)

1. For each node, assign two weights as follows:

$$weight_a = |\psi|$$
$$weight_b = \sum_{i=1}^{n} 2^i \, depend(\psi, i),$$

   where ψ is the function represented by the node, $|\psi|$ is the number of 1's in its truth table, n is the number of variables, and

$$depend(\psi, i) = 1 \quad \text{if } \psi \text{ depends on } x_i$$
$$= 0 \quad \text{otherwise.}$$

2. Select a node v_0 from SBDD0 and a node v_1 from SBDD1. For each pair of nodes (v_0, v_1) that have the same values for both weight_a and weight_b, check if they represent the same function. If so, remove the node v_i (not the subtree) that has fewer successor nodes that are shared by other parts of the SBDD. All edges leading to the eliminated node v_i, are redirected to the other node v_{1-i}.
3. Remove un-referenced nodes (e.g., a node may have no incoming edges because its predecessors were eliminated in Step 2).
4. Repeat Steps 2 and 3 until all pairs of nodes are considered.

The overall algorithm is as follows.

Algorithm 4.2: (Total Algorithm)

1. Apply Algorithm 3.2 to form a bi-partitioned SBDD.
2. Apply Algorithm 4.1 to share as many nodes as possible across the two BDDs.

5. Experimental Results

5.1 Performance of Heuristic Method

We implemented Algorithms 4.2 for many benchmark functions [23]. Table 1 lists the functions for which a bi-partitioned SBDD yielded fewer nodes than its monolithic SBDD. We applied Algorithm 3.2 to 69 functions of which 19 resulted in a reduction. In these cases, node sharing was not performed (i.e. Algorithm 3.2 only was applied). In the case of apex7, a reduction of 22.8% was achieved.

Table 2 lists the functions where the non-terminal nodes were reduced by node sharing (i.e. Algorithm 3.2 and 4.1 were used). Unfortunately, the number of nodes reduced by Algorithm 4.1 is not so large.

We applied Algorithm 3.2 to the results of Table 1, recursively. Table 3 lists the functions where the recursive application of Algorithm 3.2 reduced the total node count. For example, for C2670, a 36.7% reduction is achieved by the recursive application of Algorithm 3.2 over a single application. The horizontal line in Table 3

Table 1 Size of bi-partitioned SBDDs (using Algorithm 3.2 only).

Name	In	Out	Monolithic	Bi-partitioned	Time (sec)
apex5	117	88	1166	1163	188.2
apex6	135	99	711	675	383.8
apex7	49	37	302	233	13.7
C499	41	32	27876	27862	2037.0
C3540	50	22	34710	34509	2672.7
C880	60	26	4166	4011	615.2
cps	24	109	1095	1044	23.4
ex5	8	63	342	339	2.9
exep	30	63	675	604	77.6
frg2	143	139	1117	1116	28.8
i8	133	81	1360	1353	1029.5
i10	257	224	24054	23341	27618.8
intb	15	7	608	607	1.4
jbp	36	57	467	444	2.1
rckl	32	7	198	188	1.1
signet	39	8	1472	1320	12.8
t2	17	16	145	137	0.5
too_large	38	3	329	323	2.3
x2dn	82	56	243	220	36.7

IBM PC/AT compatible, PentiumIII 1GHz, Linux 2.2.16

distinguishes between functions where a single application decreases (above) or increases (below) the node count. Thus, for functions below the line, there is an increase initially in the number of nodes followed by a decrease, as Algorithm 3.2 is recursively applied.

Table 2 Number of non-terminal nodes reduced by node sharing.

Name	In	Out	Node reduction
apex5	117	88	8
apex6	135	99	1
apex7	49	37	14
C499	41	32	2
C880	60	26	1
C3540	50	22	17
cps	24	109	1
frg2	143	139	8
i8	133	81	21
i10	257	224	234
intb	15	7	6
rckl	32	7	3
signet	39	8	3

Table 3 Sizes of SBDDs after recursive application.

Name	In	Out	Monolithic	Bi-partitioned	
				Once	Recursive
C3540	50	22	34710	34509	34460
C880	60	26	4166	4011	4004
exep	30	63	675	604	550
i10	257	224	24054	23341	19479
intb	15	7	608	607	567
signet	39	8	1472	1326	1222
x2dn	82	56	243	220	218
C1908	33	25	7456	7707	7373
C2670	233	140	2847	2849	1801
C7552	207	108	2945	3307	2811
des	256	245	3972	4003	3862
ibm	48	17	426	427	332

5.2 Comparison of Heuristic and Exact Method

To see the quality of the bi-partitions obtained by Algorithm 3.2, we compared Algorithm 3.2 with an exhaustive method. The exhaustive method produced all the partitions of the outputs. Table 4 compares the sizes of BDDs for benchmark functions by using Algorithm 3.2 and the exhaustive method [10]. *Alg3.2* denotes the size of bi-partitioned BDDs obtained by Algorithm 3.2; *Max* denotes the maximum size of bi-partitioned BDDs over all partitions; *Min* denotes the minimum size of bi-partitioned BDDs over all partitions; *Average* denotes the average size of bi-partitioned BDDs for all the partitions.

It is noted that the heuristic, Algorithm 3.2, does not always find the optimal bi-partition, although it is better than the average in 9 out of 10 functions. This heuristic can be improved by allowing it to search more bi-partitions, for example, randomly, as in simulated annealing. Also, improved results should occur if partitions with two or more parts are allowed. Table 4(a) shows the case where the BDDs were optimized by an exact method [10]. In this method, essentially all 2^{m-1} bi-partitions and all $n!$ variable orderings are considered. Because the functions in Table 4(a) have relatively few variables and few outputs, such an algorithm completes in a reasonable time. However, for larger functions, a fast heuristic must be used. Table 4(b) shows the case where the BDDs were optimized by a heuristic method [14]. Unfortunately, bi-partitioned BDDs are often larger than monolithic ones. Table 4(b) shows that Algorithm 3.2 obtained the minimum solution in five out of six functions.

Table 4 illustrates the advantage bi-partitioning offers when the node count reduction is small, as dis-

Table 4 Comparison of the heuristic method and exact method.

(a) When BDDs are minimized by an exact algorithm [10].

Name	In	Out	Monolithic	Bi-partitioned				
				Alg3.2		Max	Min	Average
alu2	10	8	70	75	(39/36)	87	69	80.2
clip	9	5	99	105	(82/23)	113	97	106.7
ex1010	10	10	1423	1493	(193/1300)	1577	1486	1560.2
ex7	16	5	91	92	(55/37)	94	92	93.1
intb	15	7	608	595	(367/228)	636	560	601.2
max512	9	6	184	192	(122/70)	210	189	200.0
newtpla	15	5	54	55	(13/42)	61	55	57.5
t3	12	8	66	71	(14/57)	84	69	77.7
t4	12	8	44	51	(14/37)	53	46	49.5
x2	10	7	43	44	(18/26)	50	44	47.1

(b) When BDDs are minimized by a heuristic algorithm [14].

Name	In	Out	Monolithic	Bi-partitioned				
				Alg3.2		Max	Min	Average
i3	132	6	139	140	(40/100)	140	140	140.0
rckl	32	7	198	188	(105/83)	216	188	209.2
signet	39	8	1472	1320	(583/737)	1478	1316	1377.4
vg2	25	8	90	102	(89/13)	134	102	127.0
x1dn	27	6	139	140	(50/90)	174	140	164.0
x9dn	27	7	139	140	(49/91)	189	140	177.8

cussed in the Introduction. Even if the bi-partitioned SBDD has more nodes than the corresponding monolithic SBDD, it may be more advantageous to implement the bi-partitioned SBDD especially if there no or few shared nodes. Such a situation allows a divide-and-conquer approach to logic implementation, since layout, is simplified with two small circuits verses one large one. Table 4 shows the number of nodes in each of the two parts of the partitioned SBDD resulting from the application of Algorithm 3.2. For example, for alu2, the two parts have 39 and 36 nodes. There is no node sharing, so that these two SBDDs can be laid out and placed separately. In the case of ex1010, there is a large disparity in the size of the two parts, 193 and 1300 nodes. For this case, partitioning has a smaller advantage.

In Tables 1–4, our SBDDs do not use complemented edges.

6. Conclusions and Comments

In this paper, we showed a new method to represent a multiple-output function, partitioned SBDDs. Partitioned SBDDs represent a multiple-output function by a set of SBDDs, where each SBDD is optimized independently. The partitioned SBDD is more canonical than partitioned BDDs and free BDDs (FBDDs). We developed a heuristic bi-partition algorithm for SBDDs, and showed cases where the total number of nodes in bi-partitioned SBDDs are smaller than in monolithic SBDDs.

The advantages of partitioned SBDDs are

1) For each group of outputs, the orderings of the input variables are the same. So we can use well-developed tools for SBDDs [21].
2) When no node sharing among SBDDs is allowed, they can be evaluated in parallel. This may be appropriate for logic simulation applications [18], where it is important to partition the BDD into pieces that can be tractably processed.
3) When there is no sharing of nodes among SBDDs the layout can be done separately. This improvement in flexibility can be significant [9].

In this paper, we have focused on hardware synthesis. However, our results can also be applied to

1) Software synthesis [2], [18]. Replace each non-terminal node by an *if then else* statement, forming a branching program for F. In such applications, minimizing the number of nodes in the SBDD minimizes the size of the program required in the implementation. Also, partitioning an SBDD allows the corresponding programs to be developed separately perhaps by different individuals.
2) Verification [6], [12], [13]. In verification, a monolithic BDD may be too large to be stored in a computer memory. So, in this case, the BDD is partitioned into smaller BDDs that are analyzed sequentially.

In hardware synthesis, one seeks as useful partitions that have the property

$$size(SBDD, F_1) + size(SBDD, F_2) < size(SBDD, F).$$

However, for verification, the criteria for usefulness is different. Each SBDD is stored in computer memory one at a time, and the partition is used to reduce the peak memory size. In such a case, the bi-partitions are used to reduce $\max\{size(SBDD, F_1), size(SBDD, F_2)\}$.

Partitioned BDDs are considered in [12]. Their application is verification, in which case, extremely large BDDs are needed. Partitioning is a means of reducing BDD size so that each part fits into memory. Their experimental results show that the total sizes over all parts of a partitioned BDD are less than the size of the original un-partitioned BDD in 13 out of 20 benchmark functions. That is, partitioning results in a reduction in size in 65% of the benchmark functions.

Acknowledgments

This work was supported in part by a Grant Japan Society for the Promotion of Science (JSPS), and the Takeda Foundation.

References

[1] P. Ashar and S. Malik, "Fast functional simulation using branching programs," ICCAD'95, pp.408–412, Oct. 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. Comput.-Aided Des. Integrated Circuits & Syst., vol.18, no.6, pp.834–849, June 1999.
[3] S.D. Brown, R.J. Francis, J. Rose, and Z.G. Vranesic, Field Programmable Gate Arrays, Kluwer Academic Publishers, Boston, 1992.
[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] G. Cabodi, S. Quer, C. Meinel, H. Sack, A. Slobodov, and C. Stangier, "Binary decision diagrams and the multiple variable order problem," Proc. 1998 IEEE/ACM Int. Workshop on Logic Synthesis, IWLS-98, pp.346–352, Lake Tahoe, CA, 1998.
[6] G. Cabodi, P. Camurati, and S. Quer, "Improving the efficiency of BDD-based operators by means of partitioning," IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., vol.18, no.5, pp.545–556, May 1999.
[7] J. Gergov and C. Meinel, "Efficient analysis and manipulation of OBDDs can be extended to FBDDs," IEEE Trans. Comput., vol.43, no.10, pp.1197–1209, Oct. 1994.
[8] W. Gunther and R. Drechsler, "Minimization of free BDDs," Proc. Asia and South Pacific Design Automation Conference, pp.323–326, Jan. 1999.

[9] P. Kudva, A. Sullivan, and W. Dougherty, "Metrics of structural logic synthesis," Proc. 2002 IEEE/ACM Int. Workshop on Logic Synthesis, IWLS-02, pp.1–6, New Orleans, LA, 2002.
[10] N. Ishiura, H. Sawada, and S. Yajima, "Minimization of binary decision diagrams based on exchange of variables," ICCAD-91, pp.472–475, Nov. 1991.
[11] 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 Conference, pp.52–57, June 1990.
[12] A. Narajan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli, "Partitioned ROBDDs—A Compact, canonical and efficient manipulable representation of Boolean functions," Proc. IEEE/ACM ICCAD'96, pp.547–554, San Jose, CA, USA, Nov. 1996.
[13] A. Narajan, A. Isles, J. Jain, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Reachability analysis using partitioned-ROBDDs," Proc. IEEE/ACM ICCAD'97, pp.547–554, San Jose, CA, USA, 1997.
[14] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," Proc. IEEE International Conference on Computer-Aided Design, pp.42–47, Santa Clara, CA, Nov. 1993.
[15] T. Sasao, Switching Theory for Logic Synthesis, Kluwer Academic Publishers, 1999.
[16] T. Sasao and M. Fujita, eds., Representations of Discrete Functions, Kluwer Academic Publishers, 1996.
[17] T. Sasao and J.T. Butler, "A method to represent multiple-output switching functions by using multi-valued decision diagrams," International Symposium on Multiple-Valued Logic, pp.248–254, Santiago de Compostela, Spain, May 1996.
[18] T. Sasao, M. Matsuura, and Y. Iguchi, "Cascade realization of multiple-output function and its application to reconfigurable hardware," International Workshop on Logic and Synthesis, pp.225–230, Lake Tahoe, June 2001.
[19] C. Scholl, R. Drechsler, and B. Becker, "Functional simulation using binary decision diagrams," ICCAD'97, pp.8–12, Nov. 1997.
[20] R.S. Shelar and S.S. Sapatnekar, "Recursive bipartitioning of BDDs for performance driven synthesis of pass transistor logic circuits," ICCAD 2001, pp.449–452, Nov. 2001.
[21] F. Somenzi, "CUDD: CU decision diagram package," Public Software, University of Colorado, Boulder, CO, April 1997. http://vlsi.colorado.edu/ fabio/.
[22] M. Tachibana, "Heuristic algorithms for FBDD node minimization with application to pass-transistor-logic and DCVS synthesis," Proc. SASIMI Workshop, pp.96–101, Nov. 1996.
[23] S. Yang, Logic synthesis and optimization benchmark user guide version 3.0, MCNC, Jan. 1991.

Munehiro Matsuura was born on January 1, 1965 in Kitakyushu City, Japan. He studied at the Kyushu Institute of Technology from 1983 to 1989. 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.

Tsutomu Sasao received the B.E., M.E., 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, IBM T.J. Watson Research Center, Yorktown Height, NY and the Naval Postgraduate School, Monterey, CA. Now, he is a Professor of Department of Computer Science and Electronics, as well as the Director of the Center for Microelectronic Systems 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 9 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, 2001 respectively. He has served 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, and Distinctive Contribution Awards from IEEE Computer Society MVL-TC in 1987 and 1996 for papers presented at ISMVLs, and Takeda Techno-Entrepreneurship Award in 2001. He has served an associate editor of the IEEE Transactions on Computers. Currently, he is the Chairman of the Technical Committee on Multiple-Valued Logic, IEEE Computer Society. He is a Fellow of the IEEE.

Jon T. Butler received the B.E.E. and M.Engr. degrees from Rensselaer Polytechnic Institute, Troy NY in 1966 and 1967. He received the Ph.D. degree from The Ohio State University, Columbus, OH in 1973. Since 1987, he has been a professor at the Naval Postgraduate School, Monterey, CA. From 1974 to 1987, he was at Northwestern University, Evanston, IL. During that time, he served two periods of leave at the Naval Postgraduate School, first as a National Research Council Senior Postdoctoral Associate (1980–1981) and second as the NAVALEX Chair Professor (1985–1987). He served one period of leave as a Foreign Visiting Professor at the Kyushu Institute of Technology, Iizuka, Japan. His research interests include logic optimization and multiple-valued logic. He has served on the Editorial Boards of the IEEE Transactions on Computers, Computer, and IEEE Computer Society Press. He has served as an Editor-in-Chief of Computer and IEEE Computer Society Press. He has received the Award of Excellence, the Outstanding Contributed Paper Award, and a Distinctive Contributed Paper Award for papers presented at the International Symposium on Multiple-Valued Logic. He has received the Distinguished Service Award, two Meritorious Service Awards, and nine Certificates of Appreciation for service to the IEEE Computer Society. He is a Fellow of the IEEE.

Yukihiro Iguchi was born in Tokyo, and received the B.E., M.E., and Ph.D. degrees in electronic engineering from Meiji University, Kanagawa Japan, in 1982, 1984, and 1987, respectively. He is now an associate professor of Meiji University. His research interest includes logic design, switching theory, and reconfigurable systems. In 1996, he spent a year at Kyushu Institute of Technology.