計算機アーキテクチャ　98－4
設　計　自　動　化　65－4
(1993. 1. 21)

二次記憶に格納された大きな共有二分決定グラフを扱うための幅優先アルゴリズム

越智 裕之[†]　　安岡 孝一[‡]　　矢島 脩三[†]

[†] 京都大学 工学部 情報工学教室
[‡] 京都大学 大型計算機センター

〒606-01 京都市左京区吉田本町

あらまし

共有二分決定グラフは論理照合、タイミング検証、順序機械の設計検証、テスト生成、論理合成などの分野で幅広く利用されている。しかし、扱う論理関数が複雑になると計算機の主記憶容量の制約を受けるため実用面で大きな障害となっていた。本稿では計算機の二次記憶上に格納された大きな共有二分決定グラフを効率良く操作するための幅優先アルゴリズムを提案する。ワークステーション Sun SPARC Station 2 上に実現して 500MB のハードディスク領域を利用して行なった実験では、例えば 14 ビット乗算器の回路記述からこれの全ての出力論理関数を表す共有二分決定グラフを約 7 時間で生成することができた。本手法により、論理回路設計支援システムの可能性が大きく広がるものと期待される。

和文キーワード　二分決定グラフ、二次記憶、論理関数、設計検証

A Secondary Storage Oriented Breadth-First Algorithm for Manipulating Very Large SBDD's

Hiroyuki OCHI[†], Koichi YASUOKA[‡] and Shuzo YAJIMA[†]

[†] Department of Information Science
Faculty of Engineering
[‡] Data Processing Center

Kyoto University, Kyoto 606-01, Japan

Abstract

Boolean function manipulators based on Shared Binary Decision Diagrams (SBDD's) are widely utilized in formal design verification, etc. However, in many applications we have to give up to prove large-scale problems due to the limitation of the size of the main memory to store SBDD's. We propose a breadth-first algorithm for efficient manipulation of very large SBDD's stored in the secondary storage. This proposed algorithm is implemented on the workstation Sun SPARC Station 2. Using 500MB hard disk space, an SBDD for 14-bit multiplier is constructed in 7 hours from a circuit description. The developed technique for SBDD manipulation is expected to enable us much larger and more complex design.

英文 key words　Binary decision diagram, secondary memory, Boolean function, design verification

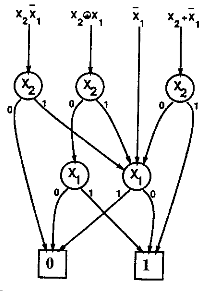
1 Introduction

Boolean function manipulation is one of the most essential operations in various applications of Computer-Aided Design (CAD) of digital systems. Because the efficiency of Boolean function manipulation is closely connected with the representation of Boolean functions, various representations of Boolean functions have been proposed. A Shared Binary Decision Diagram (SBDD) is a graph representation of Boolean functions [1][2]. Because of its excellent properties to realize efficient Boolean function manipulation, SBDD's are widely used in various applications, including formal design verification, test generation, logic synthesis and so on.

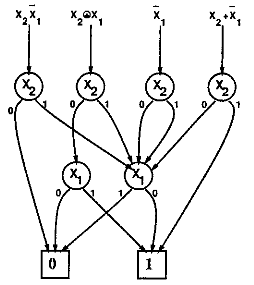
At present, SBDD manipulators are, in most cases, implemented on workstations [3][4]. The recent progress in VLSI technologies, however, requires to manipulate larger-scale Boolean functions. The maximum size of the SBDD's which can be manipulated on workstations is limited by both the required time for manipulating SBDD's and the required memory to store SBDD's. In order to reduce the computation time, the use of parallel machines or connection machines [5] and the use of vector supercomputers [6] have been proposed. However, yet in many applications we have to give up to prove large-scale problems due to the limitation of the size of the main memory to store SBDD's rather than the computation time. In order to reduce the size of SBDD's, attributed edges have been proposed [3][4]. Ordering of the input variables has been also studied by many researchers.

In this paper, the use of the secondary memory, such as the hard disk of workstations or the semiconductor extended storage of vector supercomputers, is considered in order to manipulate very large SBDD's which is not able to store in the main storage. In contrast to conventional algorithms that are based on depth-first manipulation which causes the random access of the memory, this proposed algorithm is based on breadth-first manipulation; SBDD's are manipulated level-by-level. The nodes of a level are recalled from the secondary memory in one lot, then the operations for the nodes of the level are performed in the main memory, and the nodes of the level are stored to the secondary memory together. This breadth-first algorithm is effective to reduce the overhead due to the access to the secondary memory, because it requires small number of the accesses for large data blocks in the contiguous space of the secondary memory rather than frequent random accesses for small data blocks scattering in the space of the secondary memory.

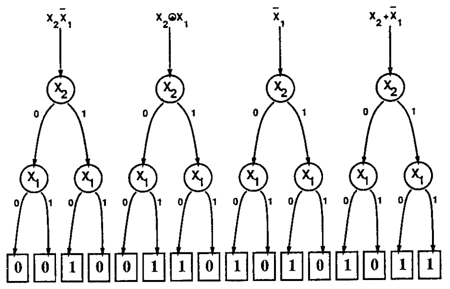
We implemented and evaluated the proposed algorithm on the workstation Sun SPARC Station 2 with 64 mega byte main memory and SCSI hard disk. More than 28 million nodes can be allocated within 500 mega byte virtual memory space, and an SBDD representing the primary outputs of the 14bit multiplier can be constructed from the circuit description in about 7 hours. If the conventional SBDD manipulator manipulates the SBDD stored in the virtual memory space which is much larger than physical main memory size, it is estimated that it takes about 35 times longer time than our manipulator.



(a) A Shared Binary Decision Diagram



(b) Quasi-Reduced SBDD



(c) Binary Decision Trees

Fig. 1. An SBDD, a Quasi-Reduced SBDD and Binary Decision Trees

2 Preliminaries

2.1 SBDD

A *Shared Binary Decision Diagram (SBDD)* is a representation of Boolean functions using an acyclic directed graph [1][2]. An example of an SBDD is shown in Fig. 1(a). This graph represents four Boolean functions corresponding to four *root-edges*. The node pointed to by the root-edge of a Boolean function is referred to as *root-node* of the Boolean function. There are (at most) two sink nodes, *leaf-nodes*, which are labeled with 0 and 1. Every node other than leaf-nodes, called *internal-node* or simply *node*, is labeled with a Boolean variable. Every node has exactly two outgoing edges. They are labeled with '0' and '1'. They are called *'0' edge* and *'1' edge*, respectively.

In this paper, we consider *quasi-reduced SBDD's* (Fig. 1(b)) in order to allow the level-by-level manipulation. Quasi-reduced SBDD is defined as the directed acyclic graph obtained from the binary decision trees (Fig. 1(c)) by repeating the following transformations until they are not applicable.

- Share isomorphic sub-graphs.
- Delete every node both of whose '0' edge and '1' edge point to the same leaf-node.

If all those nodes both of whose '0' edge and '1' edge point to the same internal-nodes are deleted, then an SBDD is obtained. Note that no Boolean variable appears more than once in every path of an SBDD, and the variables appear in a fixed order in all the paths of an SBDD. An integer number, called *level*, is assigned to every Boolean variable with respect to the ordering of the variables in an SBDD. This assignment corresponds to the ordering so that a variable nearer to the leaf-nodes has a smaller number. We denote the variable with level i as x_i. The above statements are also true of quasi-reduced SBDD's. Quasi-reduced SBDD's are different from SBDD's in the following points;

- Every '0' edge and '1' edge of a level i node points to either a level $i-1$ node or a leaf-node.
- Root-nodes which are externally referred to by users have the common level, called *level_max*, except the root nodes which represent 0 or 1.
- There may be nodes whose '0' edge and '1' edge point to the same internal-node. We call such nodes as *redundant-nodes.*

SBDD's and quasi-reduced SBDD's have following excellent properties:

- Canonical, i. e. there are no two root-edges of a graph which point to the different nodes and yet represent the same Boolean function [1][2].
- The size of the graph is feasible for many of the practical Boolean functions.
- The manipulations for various operations on Boolean functions represented by an SBDD (or a quasi-reduced SBDD) can be done in time proportional to the number of the nodes of the graph [2]. Therefore Boolean functions represented in the feasible size can be manipulated in feasible time.
- The equivalence of two Boolean functions can be tested simply by comparing the root-edges corresponding to the functions.

We denote the sub-function of Boolean function f, which is obtained by substituting 0 (1) for the variable x_i, as $f(x_i = 0)$ $(f(x_i = 1))$, or simply f_0 (f_1) if x_i is obvious from the context.

2.2 A Conventional Algorithm for Manipulating SBDD's

The principal tasks of Boolean function manipulators are

(1) comparison of two Boolean functions,

(2) the unary operation for a Boolean function (i. e., NOT), and

(3) binary operations for Boolean functions, including AND, OR, EX-OR, and so on.

If the Boolean functions are represented by an SBDD, (1) can be achieved simply by comparing two root-edges of the given functions. (2) is also easily realized if output inverters [3] are employed. The major focus of the remaining sections, therefore, is placed on algorithms to achieve (3).

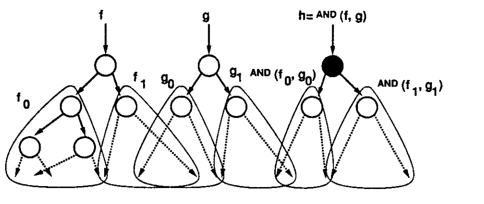


Fig. 2. Conventional Recursive Algorithm

For example, let us consider a conventional recursive algorithm [3][4] for generating the graph that represents the Boolean function h=AND(f, g), where f and g are Boolean functions represented by a given SBDD with two root-edges e_f and e_g. We denote the levels of the root-nodes of f and g as L_f and L_g, and let $L_h = max(L_f, L_g)$.

[A Conventional Algorithm for AND]

Examine the given two edges, e_f and e_g, and execute one of the following statements.

(1) If e_f or e_g point to the leaf-node 0, then return the edge pointing the leaf-node 0.

(2) If e_f (e_g) points to the leaf-node 1, then return e_g (e_f).

(3) If $e_f = e_g$, then return e_f.

(4) Otherwise, compute the root-edges of $h(x_{L_h} = 0) =$ AND $(f(x_{L_h} = 0), g(x_{L_h} = 0))$ and $h(x_{L_h} = 1) =$ AND $(f(x_{L_h} = 1), g(x_{L_h} = 1))$, recursively. Then examine the root-edges of $h(x_{L_h} = 0)$ and $h(x_{L_h} = 1)$ and execute one of the following statements.

  (5.1) If $h(x_{L_h} = 0) = h(x_{L_h} = 1)$, then return the root-edge of $h(x_{L_h} = 0)$.

  (5.2) Otherwise, generate a new root-node for h whose level is L_h and whose '0' edge and '1' edge point to the root-node of $h(x_{L_h} = 0)$ and $h(x_{L_h} = 1)$, respectively (Fig. 2).

Before generating a new node in (5.2) of the above algorithm, it should be examined whether there exists a node whose level is L_h and whose '0' edge and '1' edge point to the root-node of h_0 and h_1, respectively. If such a node exists, this old node must be used instead of generating a new node. This task is crucial for keeping SBDD canonical. For this purpose, a hash table, *node-table*, is introduced to manage all the nodes of the graph. The keys of the node-table are the level, '0' edge and '1' edge of a node.

Another hash table, *operation-result-table*, is introduced to avoid repeating the same operations. The keys of the operation-result-table are a Boolean operator (e. g. AND) and given two edges. Every time when (5) of the above algorithm is completed, the result is registered to this table. Execution time for the statement (5) is saved when the result is found in this table before executing the statement (5). This table is not only effective but also essential especially when there are many reconvergences in the sub-graphs of the given functions. The simple example is shown in Fig. 3. Let us consider the case of computing AND(f, g). According to the above algorithm, one must obtain AND$(f(x_{35} = 0), g)$ and AND$(f(x_{35} = 1), g)$. In order to obtain AND$(f(x_{35} = 0), g)$, both AND$(f(x_{35} = 0, x_{33} = 0), g)$ and AND$(f(x_{35} = 0, x_{33} = 1), g)$ are required, while in order to obtain AND$(f(x_{35} = 1), g)$, both AND$(f(x_{35} = 1, x_{33} = 0), g)$ and AND$(f(x_{35} = 1, x_{33} = 1), g)$ are required. Because $f(x_{35} = 0, x_{33} = 0)$ is equal to $f(x_{35} = 1, x_{33} = 0)$, the result of AND$(f(x_{35} = 0, x_{33} = 0), g)$ can be reused as the result of AND$(f(x_{35} = 1, x_{33} = 0), g)$ if the operation-result-table is introduced.

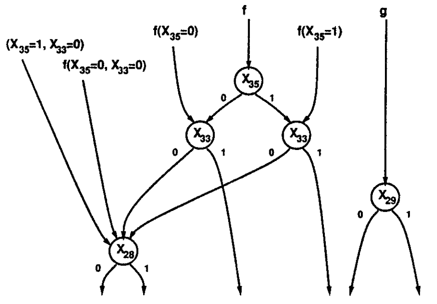


Fig. 3. Effect of the Operation-Result-Table

2.3 Secondary Storage

Today, almost all general purpose computers have secondary storages which have much larger capacity than main memories. In this paper, we assume the following devices as the secondary storages.

Hard disk of workstations

One of magnetic memory devices whose capacity is hundreds of mega bytes or more. In section 4, we will show the experimental results on the workstation Sun SPARC Station 2 (SunOS 4.1.1) with 64 mega byte main memory and SCSI hard disk drives, using the hard disk implicitly as the physical memory device of the virtual memory space supported by OS.

Fig. 4 shows the result of the following simple benchmark program for estimation of the transfer rate of the hard disk compared with that of the main memory.

```
void mwrite (mem, bytes, data)
int  *mem, bytes, data;
{
     int  i;
     for(i=0;i<bytes/sizeof(int);i++)
          mem[i]=data;
}
```

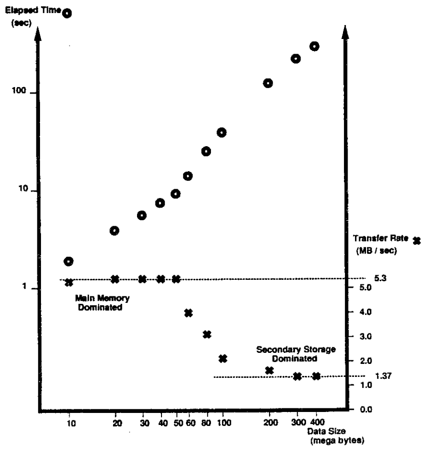


Fig. 4. Transfer Rate of Secondary Storage v.s. Main Memory

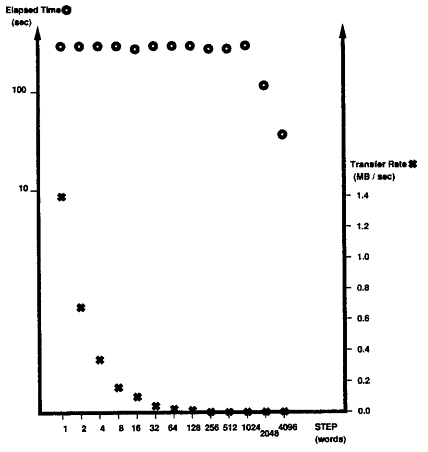


Fig. 5. Transfer Rate of Hard Disk for Sparse Data
(Array Size = 400 mega bytes)

From Fig. 4, the average transfer rate of the hard disk is approximately 1.37 mega bytes per seconds, while that of the main memory is about 5.3 mega bytes per seconds. Therefore, the transfer rate of the hard disk is about 1/4 of the transfer rate of the main memory.

Fig. 5 shows the result of the other benchmark program as follows in order to examine the relation between transfer rate and data density.

```
void mwrite(mem, step, data)
int  *mem, step, data;
{
     int  i;
     for(i=0;i<400*1024*1024/sizeof(int);i+=step)
          mem[i]=data;
}
```

From Fig. 5, elapsed time is independent of the data density when *step* is not greater than 1024 words (1 word = 4 bytes). This is because the transfer between main memory and the hard disk is performed by a block transfer of contiguous 4096 byte space. The minimum unit of the transfer between main memory and the hard disk, 4096 bytes, is called *page*. In order to keep the maximum transfer rate, every transferred page should be filled with actually used data.

Semiconductor extended storage of vector supercomputers

A secondary memory made of semiconductor memory devices, such as DRAM's. Its capacity is several giga bytes. Similar to the hard disk of workstations, the transfer between main memory and the semiconductor extended storage is performed by a block transfer of the page. The page size is one kilo bytes or more, and the access time overhead is relatively great, as discussed above.

3 A Breadth-First Algorithm for Manipulating SBDD's on the Secondary Storage

As mentioned in the previous section, the conventional algorithm for manipulating SBDD's is based on a recursive procedure (i. e. depth-first algorithm). On the memory access during the depth-first algorithm, the following can be said;

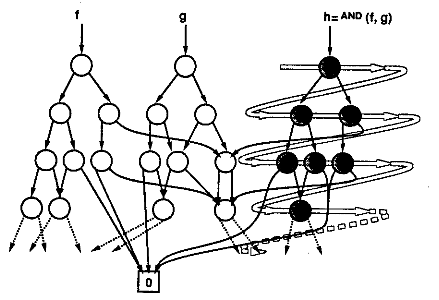


Fig. 6. Expansion Phase of the Breadth-First Algorithm

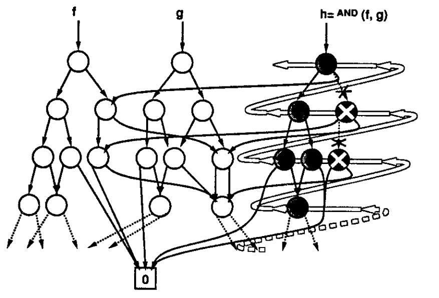


Fig. 7. Reduction Phase of the Breadth-First Algorithm

- The access to the nodes causes random access, because there are so many traversal order of the nodes for so many root-edges.
- The access to the operation-result-table and the node-table causes random access, because they are hash tables.

In this section, we propose a breadth-first algorithm for efficient manipulation of SBDD's stored in the secondary storage.

The proposed algorithm consists of two parts; an *expansion phase* and a *reduction phase*. In the expansion phase, new nodes that are sufficient to represent the resultant function are generated in a breadth-first manner from the root-node toward the leaf-nodes (Fig. 6). In the reduction phase, the nodes generated in the expansion phase are checked and the pseudo-leaf nodes and the non-unique nodes are removed in a breadth-first manner from the nodes nearby the leaf-nodes toward the root-node (Fig. 7). The nodes generated in the expansion phase are called *temporary nodes*, while the nodes which already exists are called *permanent nodes*. In order to allow the level-by-level manipulation, the proposed algorithm manipulates quasi-reduced SBDD's.

3.1 The Expansion Phase

An input for the expansion phase is a triple (op, e_f, e_g), where op is a Boolean operator to be executed, and e_f and e_g are root-edges of operand Boolean functions represented by a quasi-reduced SBDD. We refer to this triple as a *requirement*. The requirement (op, e_f, e_g) requires to compute a root-edge of the resultant function of $op(f, g)$. During the processing of the requirements of the level i, new requirements of the level $i - 1$ will be generated for computing the operations between sub-functions of the operand functions. Actually, every requirement corresponds to a procedure call in the depth-first algorithm. We introduce a queue, a *requirement queue*, for each level in order to manage these requirements, making our procedure breadth-first. (The procedure would be depth-first if we use a stack instead of the queue.)

For the given requirement (op, e_f, e_g), a new root-node is not always generated. A new node should not be generated if a node representing the result of $op(f, g)$ already exists. For example, if the result of $op(f, g)$ is found trivial (refer to (1)-(3) of the conventional algorithm), or determined by looking up the operation-result-table, a new node is not generated. In these cases, a judgement can be made immediately from e_f and e_g. There are cases, however, where the existence of the root-node of $op(f, g)$ cannot be determined until the whole graph for the sub-functions of $op(f, g)$ is constructed. In the proposed breadth-first algorithm, a temporary node is generated in such cases. We examine whether this temporary node is actually essential or not in the reduction phase.

The following procedure is executed in the expansion phase. Initially, a requirement queue is empty, and there is no temporary node.

[The Expansion Phase of the Proposed Algorithm]

(1) Put the given requirement (op, e_f, e_g) to the requirement queue of level $level\_max$.

(2) $lev = level\_max$

(3) Execute one of (3.1), (3.2) or (3.3) for every requirement in the queue of the level lev.

  (3.1) If the root-node representing the result of $op(f, g)$ is found trivial, then attach the edge pointing to the node as the result of the requirement.

  (3.2) If the root-node representing the result of $op(f, g)$ is found in the operation-result-table, attach the edge that is found in the table as the result of the requirement.

  (3.3) Otherwise, generate a new temporary node of level lev and attach the edge pointing to the temporary node as the result of the requirement. At the same time, register the edge pointing to the temporary node to the operation-result-table as the result of $op(f, g)$ and put new requirements (op, f_0, g_0) and (op, f_1, g_1), whose result will be '0' edge and '1' edge, respectively, of this temporary node, to the requirement queue of the level $lev - 1$.

(4) $lev = lev - 1$

(5) If the requirement queue of the level lev is not empty, then go to (3).

Note that the temporary nodes must be registered to the operation-result-table in the expansion phase in order to avoid repeating the same operations.

Also, note that the total number of requirements processed in the above procedure is the same as the number of procedure calls in the conventional depth-first algorithm and, thus, there is no serious increase in the computation cost. The only drawback of this algorithm is an increase of the number of nodes because of quasi-reduced SBDD.

The procedure explained so far is effective for the quasi-reduced SBDD's stored in the secondary storage if all the nodes of every level are stored together in a contiguous location in the secondary storage. We can execute the requirements of the level i, only if nodes of the level i and the level $i-1$ are in the main memory. Furthermore, if we use multiple operation-result-table, one table per a level, then we can swap out all the operation-result-tables but one from the main memory during the expansion phase.

3.2 The Reduction Phase

After the expansion phase is completed, there may be the following temporary nodes;

- *A pseudo-leaf node*: A temporary node whose '0' edge and '1' edge point to the same leaf-node.
- *A non-unique node*: A temporary node whose '0' edge and '1' edge are equivalent to those of a permanent node of the same level.

The main tasks in the reduction phase are to find pseudo-leaf nodes and non-unique nodes and to remove them. In this algorithm, these tasks are executed in a breadth-first manner from the nodes nearby the leaf-nodes toward the root-node. On the other hand, temporary nodes that are neither a pseudo-leaf node nor a non-unique node are registered to the node-table.

In practice, removed pseudo-leaf nodes and non-unique nodes of the level i remain in the *free list* of the level i as *free nodes* to be recycled in the expansion phase of the succeeding operation. The '1' edge of every removed node is used as a *forwarding pointer*, which indicates the node that takes the place of the removed node. When the '0' edge or '1' edge of a temporary node of the level $i+1$ points to a removed node of the level i, the edge is modified to point to the node pointed to by the '1' edge of the removed node before checking whether the temporary node is neither a pseudo-leaf node nor a non-unique node.

The reduction phase is formalized as follows;

[The Reduction Phase of the Proposed Algorithm]

```
(1) $lev = 1$
(2) Execute (2.1) - (2.4) for every temporary nodes of the level $lev$.
    (2.1) If its '0' edge or '1' edge points to a removed node, modify the edge so as to point to the node pointed to by the '1' edge of the removed node.
    (2.2) If its '0' edge and '1' edge point to the same leaf-node, remove the node to the free list, leaving its '1' edge pointing to the leaf-node.
    (2.3) If there is an equivalent node registered in the node-table, remove the node to the free list, setting its '1' edge to point to the node found in the node-table.
    (2.4) Otherwise, register the node to the node-table, and change the attribute of the node to "permanent" from "temporary".
(3) $lev = lev + 1$
(4) If $lev \leq level\_max$, then go to (2).
```

This procedure is also effective for the quasi-reduced SBDD's stored in the secondary storage if all the nodes of every level are stored together in a contiguous location in the secondary storage. If we use multiple node-table, one table per a level, then we can swap out all the node-tables but one from the main memory during the reduction phase.

3.3 Data Structure for the Breadth-First Algorithm

As mentioned in section 3.1 and 3.2, all nodes of a level are stored in one contiguous space of the secondary storage, and both two kinds of hash tables, operation-result-tables and node-tables, are constructed in every level. The allocated secondary storage space for the nodes of every level includes free nodes for generating new temporary nodes. While there are free nodes of a level, the nodes of the level are overwritten in the same location of the secondary storage as they were. If there is no free nodes when a new temporary node should be generated, garbage collection is performed. If there is no node to be recycled anymore, then the total number of the nodes of the level is increased by 4 times by adding new free nodes, and a new location in the secondary storage is allocated to store the greater data block. As described in [4], the necessary size for the operation-result-table and the node-table to keep the efficiency of the operation is 1/4 to the number of the nodes. When the total number of the nodes of a level is updated, then the operation-result-table and the node-table of the level are also increased in size, and all of the elements are re-hashed into the larger tables. This strategy has the following advantages in the use of secondary storages;

- The allocated spaces of the secondary storage for levels are proportional to the number of the nodes of the levels. It optimizes the utilization of the space of the secondary storage.
- The data density, i. e. the number of the actually used nodes per the number of allocated nodes, is kept high during the manipulation. It is crucial to keep the data density to reduce the overhead of the access of the secondary memory, as mentioned in section 2.3.

3.4 Parallelization of Multiple Operations

If multiple requirements of Boolean operations are given simultaneously, then they can be processed together by putting them to the requirement queue of the level *level_max* at the initial step of the expansion phase of the proposed algorithm (Fig. 8). This technique is effective for parallel implementation of SBDD manipulators, because it extends the parallelism of the process [5]. This technique is even more effective for our algorithm to use the secondary storage. The number of the access for the secondary storage depends mainly on the repetition of the expansion phase and the reduction phase. Therefore, operations should be given as many as possible simultaneously in order to reduce the access time of the secondary storage.

In the case of such application as construction of an SBDD for a given Boolean formula or a given circuit description, multiple operations can be evaluated together whose maximum levels in the parse tree or in the circuit diagram are the same [5].

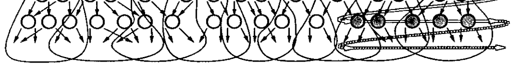


Fig. 8. Processing Multiple Operations Simultaneously

Table 1. Bechmark Results of our Manipulator

circuit	#in	#out	#node #used	#node #red.	#node #alloc	time (sec) elapsed	time (sec) CPU (user)	#I/O
mult8	16	16	11,345	208	103,023	2.459	2.30	0
mult9	18	18	31,314	336	237,165	6.640	6.22	0
mult10	20	20	87,387	567	666,219	18.438	17.20	0
mult11	22	22	241,084	959	1,661,289	51.386	48.55	0
mult12	24	24	656,710	1,650	4,547,943	154.312	135.01	523
mult13	26	26	1,794,499	2,787	13,931,877	7468.787	430.82	359,320
mult14	28	28	4,857,875	5,126	18,191,715	25945.720	1,161.22	1,230,083

Table 2. Bechmark Results of the Conventional Manipulator

circuit	#node #used	in Hard Disk time (sec) elapsed	in Hard Disk time (sec) CPU (user)	in Hard Disk #I/O	within Main Memory time (sec) elapsed	within Main Memory time (sec) CPU (user)
mult8	10,669	790	2.20	36,867	0.93	0.93
mult9	30,372	2,285	7.70	108,779	2.71	2.71
mult10	85,937	6,933	22.28	318,834	7.72	7.67
mult11	238,527	24,228	72.40	1,144,632	23.37	23.33
mult12	652,307	84,833	220.95	3,818,404	80.45	80.13

4 Implementation and Evaluation

4.1 Implementation

We implemented the proposed algorithm in C language on the workstation Sun SPARC Station 2 (SunOS 4.1.1) with 64 mega byte main memory. As the secondary storage, SCSI hard disk drives are connected. 500 mega bytes of the hard disk space is allocated as the swap area, which is used as the physical storage of the virtual memory space managed by the OS. We used the 500 mega byte hard disk space implicitly under the memory management system of the OS. This is the simplest implementation of our breadth-first algorithm.

As proposed in [4], operation-result-tables are implemented as a hash-based cache in order to omit the collision chain, while the collision chain of the node tables is implemented by an additional field, *link*, of every node. The link field is also used for managing free nodes by chaining them to *avail lists*. Of course, an avail list is constructed for every level in our implementation. The link fields of temporary nodes of every level are used to construct another linked list, *temporary list*, in order to maintain the temporary nodes of the level. There are only two requirement queues, which are always resident in main memory. During the expansion phase, when the requirements of level i are executed according to one requirement queue, new requirements of level $i-1$ are generated and put to the other requirement queue. After all the requirement of level i have been done, the queue which was used as the queue of level i will be reused as the queue of level $i-2$. The required space per a node is 18.25 bytes, including the space for the hash tables. Within the 500 mega byte virtual memory space, more than 28 million nodes can be allocated.

Minato et al. have proposed several *attributed edges*, including *output inverters*, for the purpose of reducing the number of the nodes and/or the time used for the manipulation of SBDD's [3]. The output inverter is an attribute that complements the function of the subgraph that is pointed to by the edge. We employed the output inverters for the implementation of our breadth-first algorithm.

4.2 Experimental Results

We chose multipliers as benchmarks of our algorithm in order to demonstrate the manipulation of very large SBDD which is not able to store in the main memory.

Table 1 shows the experimental results of our manipulator. This table shows the required CPU time (in user mode) and the elapsed time for constructing a quasi-reduced SBDD representing all the Boolean functions of the primary outputs of every unsigned multiplier from its circuit descriptions. The ordering of the input variable employed is

$$a_{n-1} > b_0 > a_{n-2} > b_1 > a_{n-3} > b_2 > \cdots > a_0 > b_{n-1}$$

where a's and b's are the multiplicand and the multiplier, respectively, and a_0 and b_0 are the LSB of them. This ordering requires relatively small number of nodes during the process of the construction among several systematic orderings. The column *#used* shows the number of the nodes of the final quasi-reduced SBDD necessary to represent all the Boolean functions of the primary outputs of every citcuit. The column *#red.* shows the number of the redundant nodes among them (*#used* − *#red.* is the number of the nodes required for SBDD's). We can see that the number of the nodes of a quasi-reduced SBDD is almost the same as the number of the nodes of an SBDD. The column *#alloc* shows the number of the allocated nodes. *#alloc* is much greater than *#used* because the intermediate diagram representing the Boolean functions of the internal nets of the circuit is much more complex than the final diagram. From Table 1, the elapsed time for generating a quasi-reduced SBDD for 14-bit multiplier is about 7 hours. The column *#I/O* shows the number of page faults requiring physical I/O. From Table 1, multipliers up to 11 bits required no physical I/O's. This is because they can be performed within the 64 mega byte main memory. This is yet another advantage of our implementation of incremental allocation and the use of the virtual memory space managed by OS.

Table 2 shows the experimental results of the conventional depth-first algorithm. The manipulator used for these experiments is the SBDD manipulator coded by Minato which supports two kinds of attributed edges, output inverters and input inverters [3]. His SBDD manipulator does not support the incremental allocation of memory space; all the space are allocated at the initialization process. The column *in Hard Disk* shows the result obtained by allocating (virtual) memory space for 16,777,216 nodes (372 mega bytes), that is probably the least 2's power necessary to generate an SBDD for 14-bit multiplier in order to estimate the elapsed time for 14-bit multiplier (In fact, we could not make the experiments for the multiplier of 13-bit or more, because they take too long time). The column *within Main Memory* shows the result obtained by allocating 24 mega byte memory space (If more memory space is allocated, the hard disk is activated and we cannot obtain the proper result). From Table 2, the elapsed time for generating an SBDD for 12-bit multiplier according to the conventional depth-first algorithm is about 12 hours when the SBDD is stored in the hard disk.

Fig. 9 illustrates the results shown in Table 1 and Table 2. From the figure, the following estimations can be made;

- The elapsed time required by our manipulator is about 25 times greater when the diagram is stored in the hard disk than when whole diagram is stored within the main memory. As shown in the section 2.3, the ratio of the transfer rates of the main memory and the hard disk is about 4, therefore, the actually used data in the transferred pages can be estimated as about 1/6 on the average.
- For constructing a diagram for 14-bit multiplier, the elapsed time required by the conventional manipulator using hard disk is estimated to be about 10 days, which is 35 times greater than ours. The elapsed time required by the conventional manipulator is about 1,000 times greater when the diagram is stored in the hard disk than when whole diagram is stored within the main memory. The actually used data in the transferred pages can be estimated as about 1/250 on the average. This means that only 16 bytes are actually used in every page on the average, which is the size of the structure of one node.

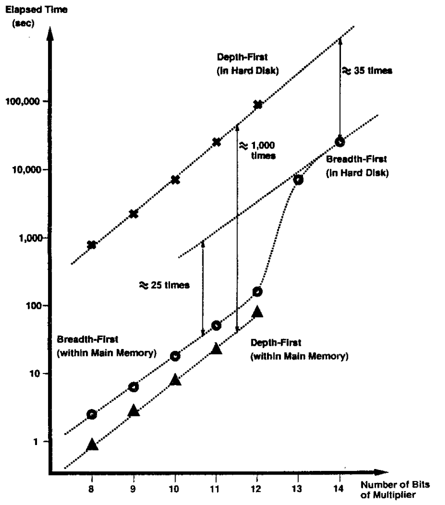


Fig. 9. Comparison of the Elapsed Time

5 Conclusion

We have proposed an efficient algorithm for manipulating an SBDD in secondary storage and shown benchmark results of the proposed algorithm on the workstation Sun SPARC Station 2 with 64 mega byte main memory and SCSI hard disk drives. An SBDD representing the primary outputs of the 14-bit multiplier can be constructed from the circuit description in about 7 hours within 500 mega byte hard disk space. Also we have shown the estimation that if the conventional SBDD manipulator manipulates SBDD in the virtual memory space that is much larger than physical main memory, it probably takes about 35 times longer time than our manipulator.

The developed technique for SBDD manipulation is expected to be utilized for various applications of CAD systems for digital systems such as formal design verification, test generation, logic synthesis and so on in order to enables us much larger and more complex design which were not possible with conventional SBDD manipulator.

The future works involved in the SBDD manipulator using the secondary storage are as follows;

- To increase the data density of transfer between secondary storage and the main memory.
- To test the developed SBDD manipulator with a semiconductor extended storage instead of a magnetic hard disk.
- To implement the proposed algorithm on a vector supercomputer with a semiconductor extended storage.

Acknowledgment

The authors would like to express their sincere appreciation to Dr. S. Kimura of Kobe University, Dr. N. Ishiura of Osaka University and Mr. S. Minato of NTT for their valuable suggestions and advices. The authors would like to thank Mr. K. Takagi and Mr. Y. Fujiyoshi of Kyoto University for their help in carrying out the experiments in section 4. Thanks are also due to all the members of the Prof. Yajima's Laboratory at Kyoto University for their valuable discussions and comments. This work was supported in part by Foundation Grant of Ministry of Ed. Japan 00041196.

References

[1] S. B. Akers: "Binary Decision Diagrams", IEEE Trans. Comput., vol. C-27, no. 6, pp. 509-516, (June 1978).

[2] R. E. Bryant: "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. Comput., vol. C-35, no. 8, pp. 677-691, (Aug. 1985).

[3] S. Minato, N. Ishiura and S. Yajima: "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation", Proc. 27th ACM/IEEE DAC, pp. 52-57, (June 1990).

[4] K. S. Brace, R. L. Rudell and R. E. Bryant: "Efficient Implementation of a BDD Package", Proc. 27th ACM/IEEE DAC, pp. 40-45, (June 1990).

[5] S. Kimura and E. M. Clarke: "A Parallel Algorithm for Constructing Binary Decision Diagrams", Proc. IEEE ICCD'90, (Sep. 1990).

[6] H. Ochi, N. Ishiura and S. Yajima: "Breadth-First Manipulation of SBDD of Boolean Functions for Vector Processing", Proc. 28th ACM/IEEE DAC, pp. 413-416, (June 1991).