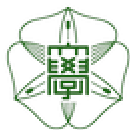


Title	Streaming BDD manipulation
Author(s)	Minato, Shin-ichi
Citation	IEEE Transactions on Computers, 51(5), 474-485 https://doi.org/10.1109/TC.2002.1004587
Issue Date	2002-05
Doc URL	http://hdl.handle.net/2115/16897
Rights	© 2002 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Type	article
File Information	IEEE-TC51-5.pdf

Instructions for use

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

Streaming BDD Manipulation

Shin-ichi Minato, *Member*, *IEEE*

Abstract—Binary Decision Diagrams (BDDs) are now commonly used for handling Boolean functions because of their excellent efficiency in terms of time and space. However, the conventional BDD manipulation algorithm is strongly based on the hash table technique, so it always encounters the memory overflow problem when handling large-scale BDD data. This paper proposes a new streaming BDD manipulation method that never causes memory overflow or swap out. This method allows us to handle very large-scale BDD stream data beyond the memory limitation. Our method can be characterized as follows: 1) it gives a continuous tradeoff curve between memory usage and stream data length, 2) valid solutions for a partial Boolean space can be obtained if we break the process before finishing, and 3) easily accelerated by pipelined multiprocessing. An experimental result demonstrates that we can succeed in finding a number of solutions to a SAT problem using commodity PC with a 64 MB memory, where as the conventional BDD manipulator would have required a 100GB memory. BDD manipulation has been considered as an intensively memory-consuming procedure, but now we can also utilize the hard disk and network resources as well. The method leads to a new approach to BDD manipulation.

Index Terms—BDD, binary decision diagram, VLSI CAD, logic design, verification, testing, data structure, algorithm, combinatorial problem.

1 Introduction

Boolean function manipulation is one of the most important techniques in digital system design and testing. Binary Decision Diagrams (BDDs) [4] are now commonly used for handling Boolean functions because of their superior efficiency in terms of time and space. A number of BDD packages (e.g., [3], [15], [23], [11], [22], [7]) have been implemented and successfully applied to many real-life problems.

In conventional BDD packages, BDD data are held in main memory. When logic operations are performed repeatedly, the size of the BDDs may dramatically increase leading to the failure (or a significant slow down) of the computation due to memory overflow. In general, we cannot know the peak BDD size beforehand, so memory overflow is a serious problem. This is a common drawback of BDD-based applications.

The cause of memory overflow is that BDD manipulation is strongly based on the hash table technique to ensure the uniqueness of each BDD node. The hash table is well supported by the random access memory and, thus, performance becomes impractical when the memory capacity is insufficient.

This paper proposes a new BDD manipulation method for processing a large number of BDD nodes beyond the hash table size. It never causes memory overflow or swap out. BDD data are accessed through the I/O stream ports. We use the main memory only as a temporary working space, where as the conventional method constructs the whole BDD data in main memory.

Some existing BDD packages (e.g., [23]) also have a function to store (and reload) the internal BDD data as a sequential file on a hard disk however, the size of the BDD data file is bounded by the main memory capacity. Our new method can handle and compute large-scale BDD data beyond the memory limitation up to the hard disk capacity, or even more than the hard disk limit through the direct connection of networked processors.

Our method can be characterized as follows: 1) it gives a continuous tradeoff curve between memory usage and stream data length, 2) valid solutions for a partial Boolean space can be obtained if we break the process before finishing, and 3) easily accelerated by pipelined multiprocessing.

This paper is organized as follows: First, we review the conventional BDD manipulation method in Section 2. We then describe our new BDD manipulation method in Section 3. We present implementation issues and experimental results in Section 4. Finally, we describe related works and concluding remarks in Section 5 and 6.

2 Conventional BDD Manipulation

Here, we briefly review the conventional BDD manipulation algorithm. BDDs are a graph representation of Boolean functions, as shown in Fig. 1a. This data structure is a reduced form of the binary decision tree, shown in Fig. 1b, where the 0- and 1-terminal nodes represent the output value $\{0,1\}$ decided by input assignments $\{0,1\}^n$. The following reduction rules [4] give a canonical form of BDD to a Boolean function under a fixed variable ordering.

- Delete all redundant nodes whose two edges point to the same node. (Fig. 2a)
- Share all equivalent subgraphs. (Fig. 2b)

- *The author is with Nippon Telegraph and Telephone, Network Innovation Laboratories, 1-1, Hikarinooka, Yokosuka-shi, 239-0847 Japan. E-mail: minato@ieee.org*

Manuscript received 23 May 2001; revised 24 Oct. 2001; accepted 11 Dec. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114187.

0018-9340/02/$17.00 © 2002 IEEE

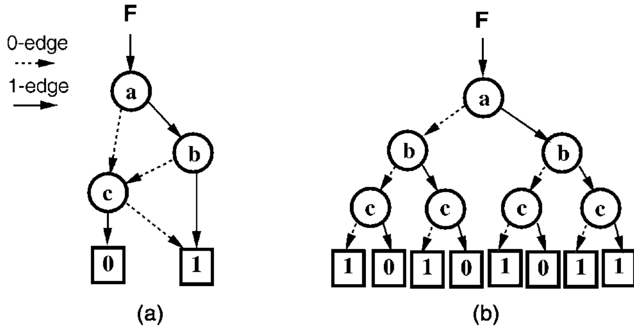


Fig. 1. BDDs for $F = ab + \bar{c}$.(a) BDD. (b) Binary decision tree.

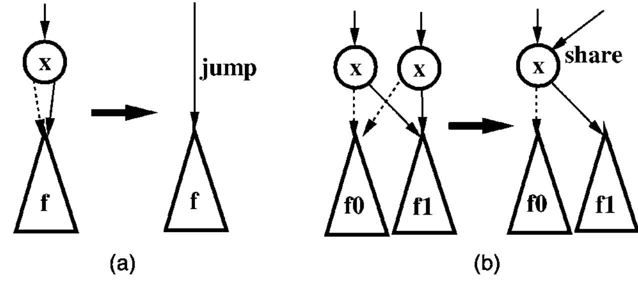


Fig. 2. Reduction rules for BDDs. (a) Node deletion. (b) Node sharing.

Most practical BDD packages follow the above reduction rules.[1]

In general, BDDs are constructed by a sequence of logic operations, starting from trivial, single-node BDDs. The binary operation algorithm [4] to generate BDD h for operation $(f \circ g)$ is based on the following expansion:

$$f \circ g = \overline{v} \cdot (f_{(v=0)} \circ g_{(v=0)}) + v \cdot (f_{(v=1)} \circ g_{(v=1)}),$$

where v is the highest ordered variable in f and g. This formula represents a new node with variable v and two subgraphs generated by suboperations $(f_{(v=0)} \circ g_{(v=0)})$ and $(f_{(v=1)} \circ g_{(v=1)})$. Repeating this expansion recursively for all the input variables eventually yields trivial operations (e.g., $f \cdot 0 = 0$, $f \oplus f = 0$, etc.), and the results are obtained. In this recursive procedure, a number of equivalent suboperations may appear. To avoid those redundant operations, the following two techniques are used;

- Unique table: use a hash table to identify all existing nodes, so as not to create duplicate nodes.
- Operation cache: a hash-based cache to store recent suboperations and the results. If this cache hits, further recursive calls are pruned.

Based on these techniques, the logic operation can be carried out in a time almost linear to the number of BDD nodes. The hash table operation is one of the keys to efficient BDD manipulation.

A typical BDD package is implemented as a set of library calls in C or C++. BDD nodes are basically defined as an array of pointers or indices in the program. The package is linked with application programs in the compilation process, and the memory block for the BDD nodes is allocated at the run time. When logic operations are performed repeatedly, the size of the BDDs may dramatically increase leading to the failure (or a significant slow down) of the computation due to memory overflow. BDD manipulation is very efficient only if memory size is sufficient.

The reason why BDDs must be handled in the main memory is that BDD manipulation is supported by the hash table technique to ensure the uniqueness of each BDD node. The hash table is well supported by the characteristics of random access memory, and thus, the performance becomes impractical when memory capacity is insufficient.

1. More exactly, this type of BDD is called Reduced Ordered BDD.

Several ideas have been proposed to increase the efficiency of memory use (e.g., [11], [22], [7]) and they offer some improvement in terms of computation time and memory requirement. On the other hand, there is endless requirement for handling large-scale Boolean functions, and a number of efforts have been devoted to manipulate huge BDDs beyond the memory limit. *Breadth-first algorithm* [17] is one solution to this problem. This algorithm horizontally slices the BDD nodes for each input variable, and manipulates them slice-by-slice. It reduces random accesses to the hard disk. In addition, there is a hybrid method [26] that uses breadth-first and depth-first manners to improve performance. However, the breadth-first algorithm still has a limitation in that at least one slice of the BDD nodes must be stored in the same hash table to keep uniqueness. When the "width of BDD" is too large, memory overflow problem still occurs.

Some works distribute the BDD data over a multiprocessing platform. This approach includes the method of partitioning BDDs by assigning $\{0, 1\}$ into a few top input variables [8], the randomized BDD node distribution method [24], and the horizontal BDD partition method which groups the input variables [19], [13]. In these methods, we can handle large-scale BDDs beyond the memory limitation of a single machine; however, they still need sufficient total memory to store all BDD nodes.

Consequently, the existing BDD packages all place some limit on BDD node number according to the memory available, and we cannot avoid memory overflow or swap out problems.

3 Streaming BDD Manipulation

In this section, we present a new algorithm of BDD manipulation based on the streaming data model.

3.1 Streaming Data Model

First, we consider the bit-stream data of the truth tables for Boolean functions, shown in Fig. 3. In this model, we can compute a logic operation bit by bit serially using no internal memory. However, the truth table representation

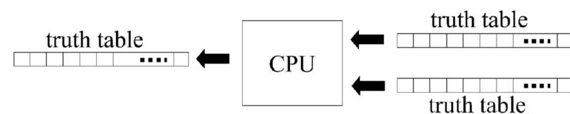


Fig. 3. Streaming truth-table computation.

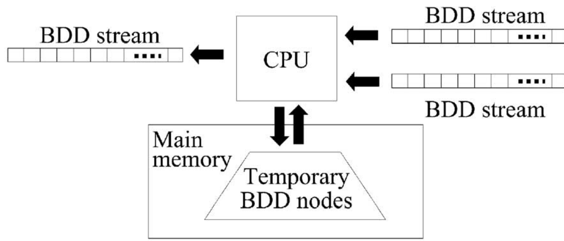


Fig. 4. Streaming BDD computation.

always requires exponential data length for an n-input function.

We then consider the streaming BDD data model, shown in Fig. 4. The serial operation of a truth table means scanning of Boolean space in a fixed order. This scanning corresponds to a depth-first traversal of a BDD. If we serialize the BDD structure data into a stream file with depth-first traversal, we can compute a logic operation using no internal memory.

3.2 Data Format

Here, we explain the basic idea of our BDD serialization method. We represent each BDD node as a pair of parentheses: `( 0-child 1-child )`. For example, the "nonshared" BDD shown in Fig. 5a is written as:

(((T1 T0)(T1 T0))((T1 T0)(T1 T1))),

where T0 and T1 are 0- and 1-terminal nodes, respectively. The nesting parentheses represent the structure of the BDD. This is just a depth-first traversal of BDD nodes when we read this stream from left to right.

We then set an identifier to each nonterminal node using the syntax: `( 0-child 1-child ): NodeID`.

The same BDD is now described as:

(((T1 T0) : N1(T1 T0) : N2) : N3((T1 T0) :
N4(T1 T1) : N5) : N6) : N7.

Next, consider the conventional "shared" BDD shown in Fig. 5b. If we traverse a shared subgraph repeatedly, the stream data length will become exponential to the number of input variables as is true for the "nonshared" BDD. Duplicative traversal can be avoided in the following way: if we find a node N_k already visited, we do not traverse the subgraph but instead say "Refer to N_k's children." Using this compression method, the BDD stream is reduced to:

(((T1 T0) : N1 N1)(N1(T1 T1)) : N2) : N3.

In this way, we can serialize the structure of a BDD as stream data whose length is linear to the original BDD data size.

We have one more reduction rule. When the same NodeID appears twice in succession in the BDD stream, we simply suppress the second appearance. For example, ((T1 T0) : N1 N1) is reduced to ((T1 T0) : N1). This corresponds to the conventional BDD reduction rule to eliminate BDD nodes whose 0- and 1-edge point to the same destination.

To produce such compressed BDD streams, we need a hash table (called unique table) as well as a conventional BDD package to identify all visited nodes. If the unique table is working well (i.e., all the visited nodes can be stored in the table), the BDD stream is just a serialized representation of the conventional BDD. The important difference is seen when the capacity of the unique table was insufficient due to memory limits. If some of the visited BDD nodes are missing from the unique table, we may sometimes fail to discover the second visit of the same node, and duplicative traversals will appear in the stream data. This means a drop in the data compression rate. Less memory will cause a further drop in the compression rate. Note that, even if we have absolutely no memory for the unique table, the BDD data streams can be computed robustly. This is a great difference from the conventional BDD packages, which cannot continue the execution.

Fig. 6 shows the syntax of our data format in a BNF-like description.[2] First, we declare the unique table size at the top of the stream. A BDD stream must start with integer `MaxID` to specify table size to the BDD manipulator. Each BDD node is identified by an integer ID from 1 to `MaxID`. We do not use symbols "N1, N2, ..." for the nonterminal nodes, but only numbers "1, 2, ...". The 0-terminal node is expressed by the special node ID "0". As we use *complement edges*, the 1-terminal is expressed as ~0, where "~" denotes the inverter to the following node. The category `StoredNode` expresses a BDD node already stored in the table, and `TempNode` is a temporary node. A `StoredNode` cannot include a `TempNode` in its children. A `NodeID` must be referred to after its registration. If a pair of parentheses encloses only one node, it indicates that the two children are equivalent. In our format, we do not need an explicit notation of the input variable for each node because the context of the parentheses indicates the corresponding variable.[3]

In Fig. 7, we illustrate some simple BDDs and their stream representations. The examples contain complemented edges. Here, we set the unique table size at 1,024 and this is declared at the top of the stream.

If the original BDD nodes do not exceed `MaxID`, the BDD stream data uniquely represent Boolean functions

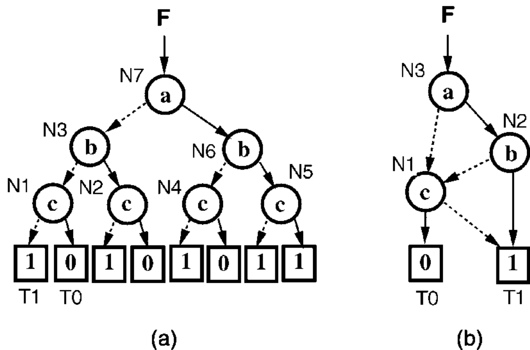


Fig. 5. Example of BDDs to be serialized. (a) Nonshared BDD. (b) Shared BDD.

2. Here, we show a plain text format for easy debugging. A binary format will be more compact.

3. Our implementation employs run-length compaction for successive parentheses: e.g., "((((((" into "(*6."

```
BDDstream  ::= MaxID Node
             | MaxID '~' Node
Node       ::= StoredNode
             | TempNode
StoredNode ::= NodeID
             | '(' StoredNode ')'
             | '(' StoredNode StoredNode '):' NodeID
             | '(' StoredNode '~' StoredNode '):' NodeID
TempNode   ::= '(' Node Node ')'
             | '(' Node '~' Node ')'
MaxID      ::= <non-negative integer>
NodeID     ::= <non-negative integer>
```

Fig. 6. Syntax of BDD data format.

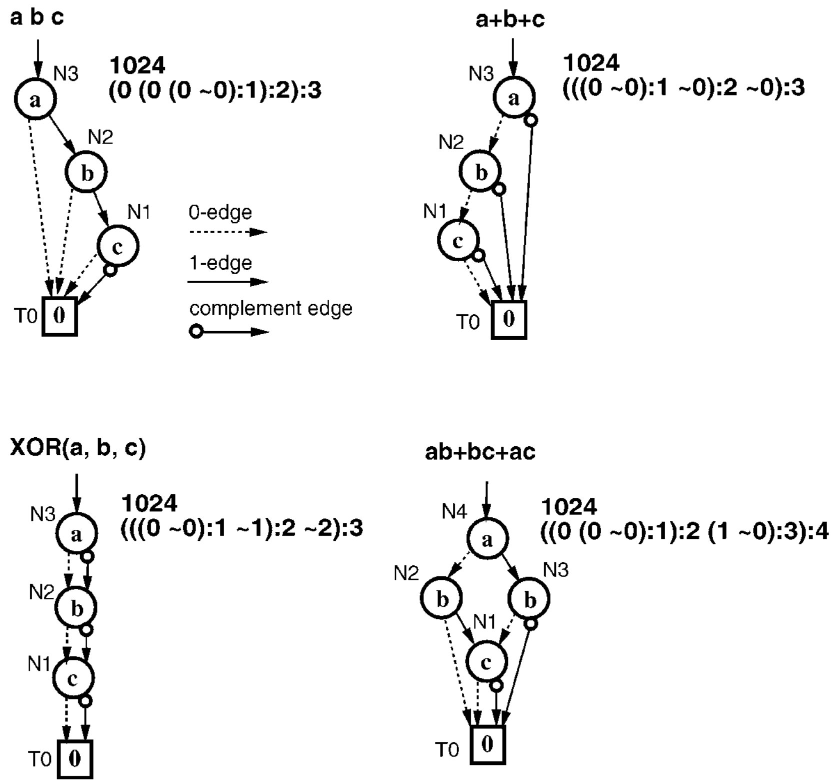


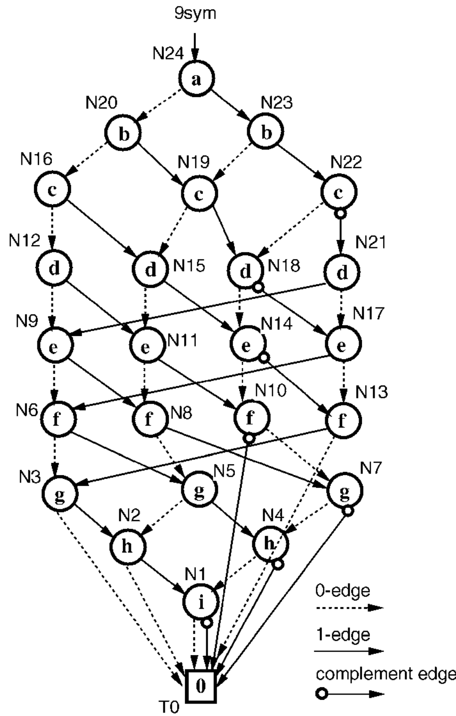
Fig. 7. BDD streams for simple functions.

under fixed variable ordering. If the table size is insufficient, the stream data may yield different representations for the same BDD. For example, Figs. 8, 9, and 10 show the BDD stream data for the same function with different `MaxID`s: 24, 20, and 10, respectively. In the first case, the node table is large enough to the complete depth first traversal of the BDD.

Table overflow occurs in the last two cases. Our traversal method allows for the recycle of a "orphan" node ID, which is not referred from other nodes. In the figures, a node label enclosed by parentheses indicates a temporary label assignment that has been reused for another node. The reader will easily understand the traversal mechanism by preparing as many numbered tokens as `MaxID`, and putting them on the BDD nodes in the figure. For example, when `MaxID`= 20, the tokens `N1` to `N20` are labeled and stored in the table normally, but we have no token for the 21st node. We then take back token `N20` at the orphan node and reuse it for the 21st node. Consequently, node `N16` newly becomes orphaned, so we reuse it the next time around. In our implementation, we prepare a recycle queue to store orphan nodes and pick up the nodes in LRU manner.

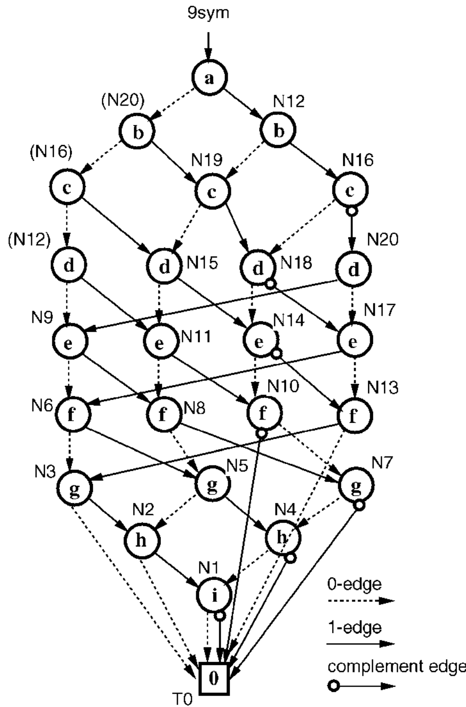
If one node label is reused, we must not store its parent node in the unique table because information would be partly lost. We call such a node `TempNode` to distinguish it from `StoredNode`. When the unique table size is smaller again, node recycling occurs more frequently, more `TempNodes` are produced, and the output data grows longer. This method gives a continuous tradeoff between memory usage and streaming data compression rate.

The choice of the orphan nodes is another technical point for efficient memory use. If we can foresee the future




```
24
(((((((0(0(0~0):1):2):3(2(1~0):4):5):6(5(4~0):7)
:8):9(8(7~0):10):11):12(11(10~(0 3):13):14):15)
:16(15(14~(13 6):17):18):19):20(19(18~(17 9):21)
:22):23):24.
```

Fig. 8. BDD stream for "9sym" (with large enough table).




```
20
(((((((0(0(0~0):1):2):3(2(1~0):4):5):6(5(4~0):7)
:8):9(8(7~0):10):11):12(11(10~(0 3):13):14):15)
:16(15(14~(13 6):17):18):19):20(19(18~(17 9):20)
:16):12).
```

Fig. 9. BDD stream for "9sym" (table size = 20).

BDD traversal, the best method is to reuse nodes that will not be referred to again by the others. However, our streaming manipulation cannot predict future information, so we use LRU policy. Anyway, when the node recycle order is fixed to a deterministic manner, our BDD stream format can uniquely represent the BDD structure under the same `MaxID`.

3.3 Logic Operation

Fig. 11 shows the internal structure of our implementation for binary logic operation. It has two temporary BDD tables for the input parts and one for the output part. The table size of each input part is automatically decided to see the top of the data stream (i.e., `MaxID`). The unique table size of the output part is set by hand (specified by a command parameter).

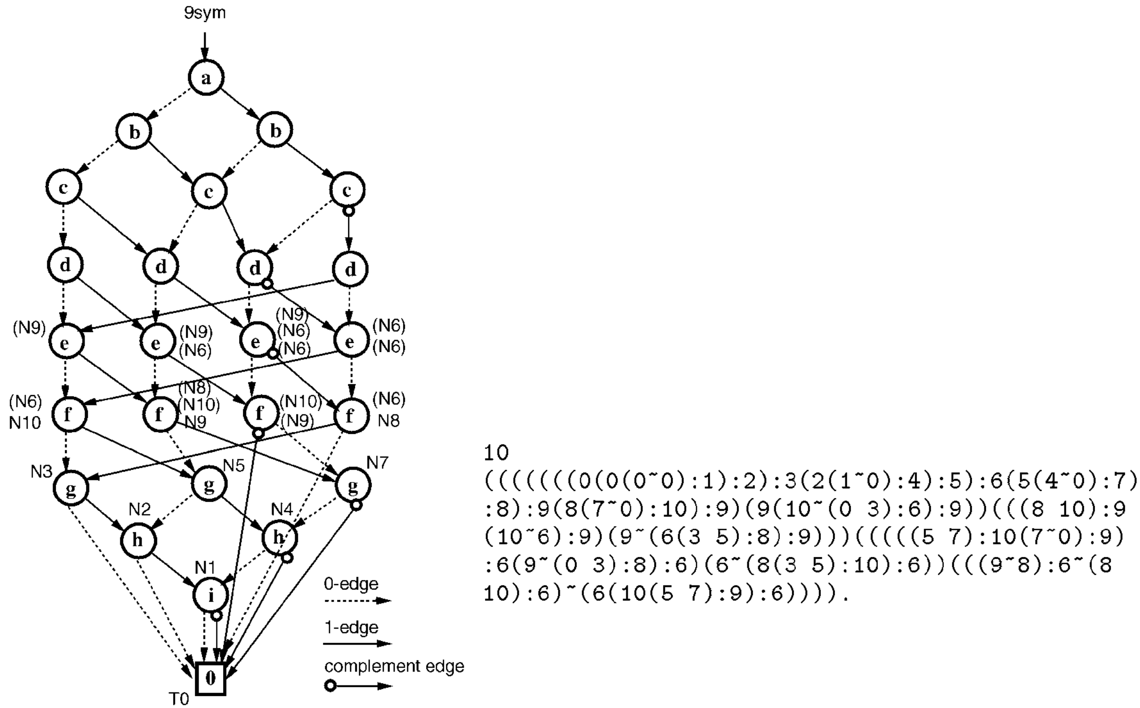


Fig. 10. BDD stream for "9sym" (table size = 10).

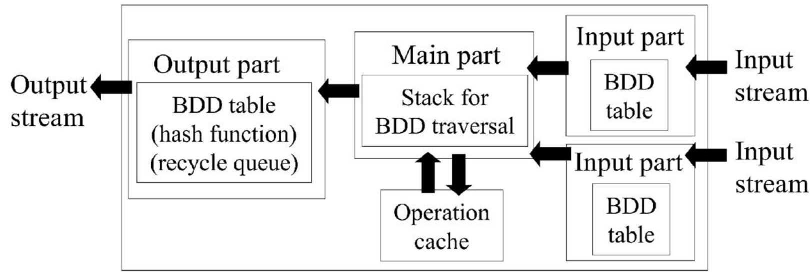


Fig. 11. Internal structure of the program.

The output part has a hash-based unique table and a recycle node queue to control memory usage. The input part does not need such a memory management system and simply forms BDD nodes on the table according to the input stream data. At first, we start parsing the input stream data and storing the BDD structure in the internal table. When the same node ID reappears in the stream, we suspend parsing and switch the traversal to the internal table. After traversal of the subgraph in the table, we resume parsing of the stream data.

The main part applies the logic operation to each pair of corresponding BDD nodes of the two input parts, and sends the result to the output part. As is done in the conventional BDD manipulation, we skip redundant suboperations by using an operation cache. This enables us to compute a logic operation in a time almost linear to the I/O data length.

We describe the pseudocodes of the conventional logic operation algorithm (Fig. 12) and our streaming algorithm (Fig. 13) below. For conciseness, here we omit the detailed codes used to arrange the parentheses and NodeIDs in the output stream. In the streaming version, "get_child" and "def_node" routines correspond to the input part functions.

```
bdd_op(op, F, G)
{   if (terminal case(op, F, G)) R ← result ;
    else if (operation cache hit(op, F, G)) R ← result ;
    else
    {   let x be the top variable of F, G ;
        F0 ← get_child(F, x, 0) ; G0 ← get_child(G, x, 0) ;
        R0 ← bdd_op(op, F0, G0) ;
        F1 ← get_child(F, x, 1) ; G1 ← get_child(G, x, 1) ;
        R1 ← bdd_op(op, F1, G1) ;
        if (R0 equals R1) R ← R0 ;
        else
        {   R ← find or add in the BDDtable(x, R0, R1) ;
            insert in the operation cache((op, F, G), R) ;
        }
    }
    return R ;
}
```

Fig. 12. Conventional logic operation algorithm.

```
bdd_op(op, F, G)
{  if (terminal case(op, F, G)) R ← result ;
   else if (operation cache hit(op, F, G)) R ← result ;
   else
   {  let x be the top variable of F, G ;
      F0 ← get_child(F, x, 0) ; G0 ← get_child(G, x, 0) ;
      R0 ← bdd_op(op, F0, G0) ;
      F1 ← get_child(F, x, 1) ; G1 ← get_child(G, x, 1) ;
      R1 ← bdd_op(op, F1, G1) ;
      def_node((x, F0, F1), F) ; def_node((x, G0, G1), G) ;
      if (R0 or R1 is TempNode) { R ← TempNode ; print "(R0 R1)" ; }
      else if (R0 equals R1) R ← R0 ;
      else
      {  R ← find or add in the output_BDDtable(x, R0, R1) ;
            if (R0 has been TempNode due to overwriting) { R ← TempNode ; print "(R0 R1)" ; }
            if (R is newly registered in the output_BDDtable) print "(R0 R1):R" ;
         insert in the operation cache((op, F, G), R) ;
      }
   }
   return R ;
}

get_child(F, x, v)
{  if (x is not the top variable of F) return F ;
   else if(F is already in the input_BDDtable(x, F0, F1)) return Fv ;
   else read from the stream input for Fv and return it;
}

def_node((x, F0, F1), F)
{  if(F0, F1 and F is given from the stream input) insert in the input_BDDtable((x, F0, F1), F) ;
}
```

Fig. 13. Streaming logic operation algorithm.

The statements for checking "output_BDDtable" and printing the result data belong to the output part.

In the streaming algorithm, we must consider the following case: When creating a new node, one of its child node data might have been invalid since it was overwritten due to node reuse. To detect such a case, we attach a timestamp to each node, and if the stamp has changed, we produce `TempNode` instead of `StoredNode`.

We also check the redundancy of the operation result. We print out the node data only in two cases:

- The node is a `Tempnode`.
- The node is newly registered in the output_BDDtable.

In this way, we can print out the NodeIDs in the same manner (depth-first traversal of a BDD) as shown in Section 3.2. For example, we sometimes see the case in which the two input streams represent complicated Boolean functions but the binary operation result always becomes zero. In this case, we do not produce "((((0 0)(0 0))((0 0)(0 0))..." since no new nodes are registered in the output_NodeTable in the operation, instead, we just print "0" upon finishing the logic operation.

Here, we have discussed on binary logic operations, but the algorithm is easily extended to support ternary (3-input) operations by adding one more input part (using not only F and G, but also, H in Fig. 13). In many cases, the use of ternary operations reduces computation time compared to a cascade of binary operations, although we need additional memory space for the extra input part. Four and more input operations are also possible in the same manner.

4 Implementation and Experimental Results

We implemented a logic operation program to manipulate BDD streams in a UNIX environment. The program, named `BDDstrm`, occupies about 2,000 lines of C code. At first, we write some trivial BDD stream files, and then repeatedly execute `BDDstrm` to construct the objective BDD streams. In the UNIX environment, we can conveniently use the pipe connection of two or more processes in a command line. For example, the sequence of logic operations "`(file1 and file2) or file3`" can be computed in the following command lines:

```
% BDDstrm AND file1 file2 > tmpfile
% BDDstrm OR tmpfile file3
```

The same operation can be done by using a pipe connection as:

```
% BDDstrm AND file1 file2 | BDDstrm OR file3
```

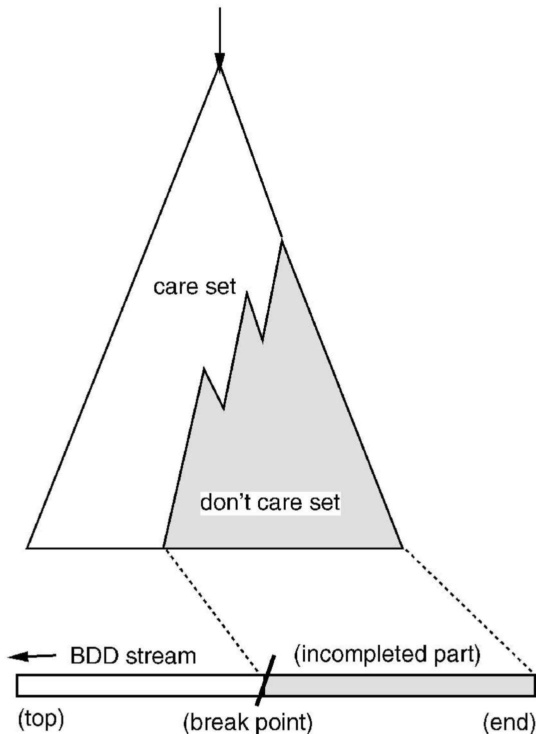


Fig. 14. Incomplete BDD stream and don't care set.

By using a pipe connection, we do not have to store the `tmpfile` in a hard disk; however, we need more memory space for the two `BDDstrm` processes.

`BDDstrm` has an option to limit the output data length. The program automatically stops at the specified limit to prevent hard disk overflow. In this case, or whenever we hit the break key before finishing, we can get an incomplete output stream data on a partial Boolean space. As shown in Fig. 14, the incomplete part can be regarded as the *don't care* set of the Boolean function. In our system, the incomplete BDD stream can be used as the input for the next logic operation. This is a great advantage of our streaming method. In addition, our program reports the ratio (%) of the care set and don't care set in the Boolean space.

As the interface to conventional BDD-based applications, we implemented an additional function in our (conventional) BDD package to print out BDD stream files by traversing in-memory BDDs. It was useful for developing, debugging, and evaluating our `BDDstrm` program. This file interface allows our streaming manipulator can utilize the results of various BDD-based program libraries.

4.1 Basic Performances

Here, we summarize the basic performance of `BDDstrm`.

- **Memory requirement:**
  - 12 Byte/node for each input BDD table.
  - 31 Byte/node for the output BDD table.
  - (about a million BDD nodes occupy a 64MB memory.)
- **Streaming data length:**
  - 5 to 15 Byte/node.
  - (about a million BDD nodes occupy a 10MB file)
- **Computation performance (I/O throughput):**
  - 0.3 to 0.5 MB/sec .
  - (about a million BDD nodes in 30 sec.)
  - on a Celeron 300A, 64MB, FreeBSD 2.6.

In terms of computation performance, our streaming method would be a few or about ten times slower than conventional BDD manipulation. This is due to the overhead of serialized manipulation. We consider that this is a reasonable trade off to be able to handle very large-scale data beyond main memory limits.

4.2 Tradeoff: Memory versus Data Length

As discussed in Section 3, the length of the BDD stream will increase when the unique table size is insufficient. To show the tradeoff curve of the data length for the table size shortage, we conducted the following experiment. First, we provide a unique table of sufficient size and counted the number of BDD nodes written in the output data. We then gradually decreased the table size to observe the growth in output data.

The results are shown in Fig. 15. "adder10" and "mult10" are a 10 bit adder and multiplier. "8queens" is the solution function of the 8-Queens problem. The other functions were chosen from MCNC'91 benchmark set. Since our program handles only single-output functions, we picked up the most (likely) complicated primary output in the circuit. In the memory sufficiency notation, 100 percent ratio means just enough to support the original BDD.

In this experiment, we can see that the tradeoff curve depends on the function. For example, "mult10," "8queens," and "cm150a" are not so sensitive to memory shortage up to 10 percent or less. This means that we can efficiently handle BDDs that are more than ten times the memory capacity. On the other hand, "parity," "c432," and "too_large" are very sensitive.

Here, we consider which types of BDDs are sensitive to the memory shortage. For example, the n-input multiplier function requires an exponential number of BDD nodes. This means that most of the graph forms a binary tree structure, and most of the BDD nodes are referred to only once. In this case, the reuse of orphan nodes has no significant effect on the stream length, so we need only $O(n)$ size of memories to traverse $O(2^n)$ BDD nodes.

A very sensitive example is presented in Fig. 16. In this case, we have a number of irrelevant input variables on the top of the BDD. Our streaming method can eliminate the redundant nodes only if all descendant nodes are stored in the unique table (i.e., they are `StoredNode`). When the memory shortage occurs, the top node becomes a `TempNode`, so an exponential number of the redundant nodes suddenly reappear. Notice that this explosion does not occur with the different variable order shown in Fig. 17. We can see that variable ordering is important not only for the original BDD size, but also for the sensitivity to memory shortage.

We should emphasize that the conventional BDD manipulation is always taxed by the memory overflow problem even if the memory sufficiency ratio is 99 percent. A great feature of our method is that it gives a continuous tradeoff between memory usage and streaming data compression rate.

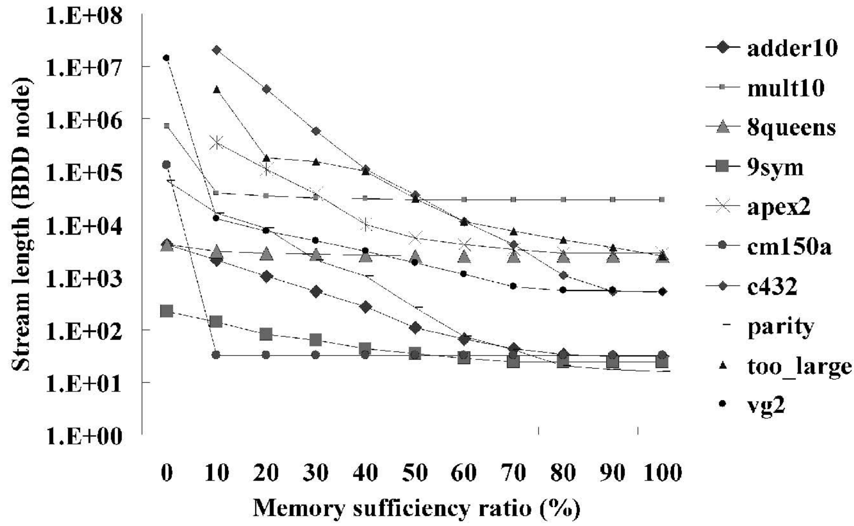


Fig. 15. Tradeoff between memory and data length.

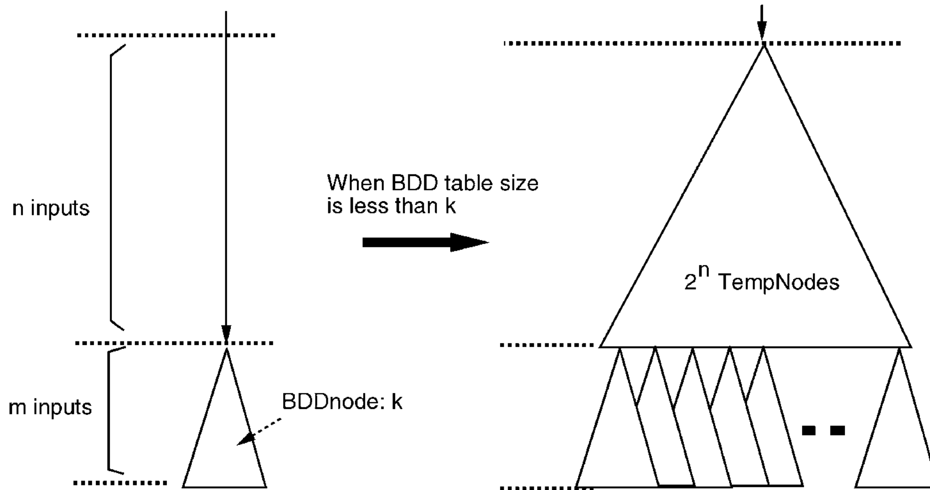


Fig. 16. A case very sensitive to memory shortage.

4.3 Experiment for Solving a SAT Problem

Many problems in LSI CAD, AI, and other fields of Computer Science can be formulated as a combinatorial problem to satisfy a set of Boolean constraints. For instance, graph coloring, the minimum flow, unate and binate covering are popular examples. SAT-based design verification/validation has also become a hot topic [21], [2], [16], [25] in recent years.

A lot of SAT procedures have been studied for many years, e.g., Davis-Putnam backtrack search [21], [16], heuristic local search [20], 0-1 linear programming [9], and BDD-based method. The BDD-based SAT procedure is a good instance of computing very large-scale BDDs, so we show here the effect of our streaming manipulation.

Several works (e.g., [12], [14], [18], [10]) have solved SAT problems using conventional BDD manipulation. Those methods first generate BDDs for the respective Boolean constraints, and then try to compute conjunction (AND operation) of all the BDDs. The final BDD represents the set of solutions that satisfy all the constraints. Unfortunately, large-scale problems often fail due to memory overflow.

We implemented a SAT-problem solver based on the streaming BDD manipulation. As shown in Fig. 18, we prepared a BDD stream file for each constraint, and compute the conjunction by a cascade of streaming BDD operations. In this system, an intermediate BDD stream represents the "candidates" of solutions satisfying the constraints processed in the upper stream. In other words, each processor filters the candidates by a constraint and, finally, the solutions are extracted.

In this system, some processors may produce duplicated nodes when memory is insufficient, however, those redundant nodes can be eliminated in the lower stream if the final result of BDD is simple. For example, if the problem is completely unsatisfiable, the simple result "0" is produced from the final processor after all data have been processed. In other words, when the first BDD node appears at the final output, we immediately know it is satisfiable. For a complicated problem, the intermediate streams may grow extremely long, and we cannot know when it will be completed. If we break the process before finishing, the incomplete output data contains a partial set of solutions. This

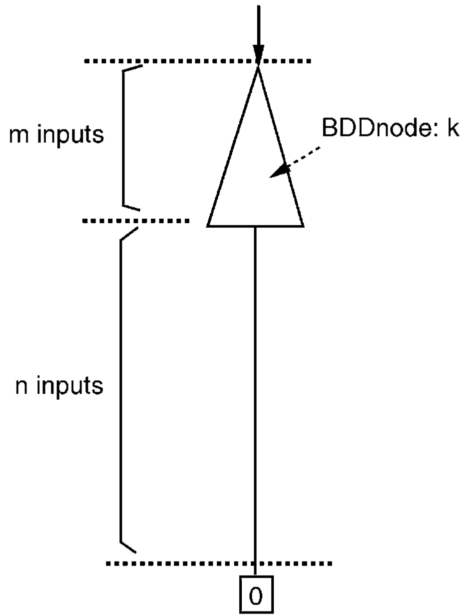


Fig. 17. Variable reordering to avoid the problem.

is a great difference from the conventional BDD manipulation, which gives no solutions when interrupted.

To evaluate the effect of our streaming manipulation, we solved several N-Queens problems, as shown in Table 1. An N-Queens problem has N^2 places on the chess-board, so we prepared N^2 Boolean variables and N^2 constraints to specify the problem. We then computed the conjunction of all Boolean constraints. The column "Prev." shows the CPU time needed to solve the same problems using conventional BDD manipulation. They are a few times faster than our streaming method for small N's, but for $N > 12$, they cannot find any solution due to memory overflow. On the other hand, our streaming method succeeded in generating all solutions up to 14-Queens.

We next conducted the same operations under the limited length of intermediate BDD streams (up to 10 MB and 1 MB). The column "#Sol." shows the number of solutions included in the final BDD stream. This result

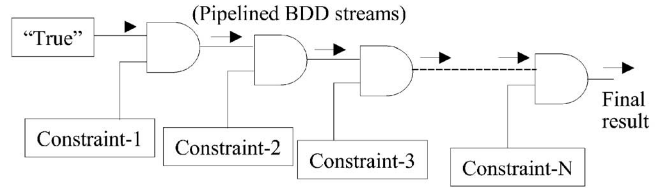


Fig. 18. Solving a SAT problem.

shows that we could compute a partial set of solutions for larger values of N in a feasible computation time, using commodity PC with 64 MB memory. This is a great advantage of our method; the conventional BDD manipulation would have required as much as 100GB of memory to execute the same operations.

In recent years, Davis-Putnam backtrack procedures have shown a remarkable improvement [16], [25], and reports show that backtracking is much faster than the BDD-based method for most SAT benchmarks. The N-Queens problem falls into the same category. However, the backtracking method is optimized for the problems specified in CNF representation. BDD-based method can solve more general logic expressions (i.e., any combinational logic expression), and it reduces the effort needed to formulate a given problem. The BDD-based method will be useful as an alternative or substitute for contemporary backtracking SAT checkers.

4.4 Pipelined Multiprocessing

If we use a single processor, we have to store an intermediate BDD stream into a hard disk at each logic operation, and read it at the next stage of operation. Using multiple networked PCs, we can deploy the logic operations across a number of PCs and concurrently execute them simply by connecting I/O ports without using any hard disk. In this way, computation can greatly be accelerated by increasing the number of PCs. The experimental results are shown in Table 2. Existing parallel BDD packages [24], [19], [13] required random data accesses between the networked processors, which is a serious bottleneck for computation. Our streaming method uses only serial data access, so remarkable acceleration is

TABLE 1
Experimental Results for N-Queens Problem

N	#Var	Prev.	Our method(Unlimited stream length)				(Limit:10MB)		(Limit:1MB)	
		CPU(s)	Peak Node	Final Node	#Sol.	CPU(s)	#Sol.	CPU	#Sol.	CPU
8	64	14.3	4,928	2,450	92	33.1	92	33.1	92	33.1
9	81	22.6	15,389	9,556	352	50.6	352	50.6	352	50.6
10	100	37.2	76,882	25,944	724	85.7	724	85.7	724	85.7
11	121	97.2	331,331	94,821	2,680	278.9	2,680	278.9	513*	161.3
12	144	395.1	1,503,336	435,169	14,200	1,214.8	9,085*	971.6	349*	218.3
13	169	MemOut	9,225,382	2,213,507	73,712	7,857.7	4,892*	1,511.3	210*	282.8
14	196	MemOut	51,638,490	12,884,133	365,596	59,479.7	2,354*	1,968.8	126*	365.9
15	225	MemOut	-	-	-	TimeOut	2,189*	2,551.1	91*	449.3
16	256	MemOut	-	-	-	TimeOut	1,307*	3,038.2	46*	517.5
17	289	MemOut	-	-	-	TimeOut	996*	3,598.1	25*	651.2

(*: Not complete set of solutions.)
(Ultra SPARC 30, 128MB Mem, 2.5GB free HD, SunOS 5.6, Time out:24h)

TABLE 2
Pipelined Multiprocessing

Solving 14-Queens		
PCs	Elapse(s)	Ratio
1	72,991	1.00
5	14,716	4.96
10	9,652	7.56
25	5,414	13.48
50	3,996	18.27
100	2,547	28.66

(Each PC: Celeron 300A
64MB Mem, 1GB free HD,
FreeBSD 2.6, 100BaseT LAN)

achieved. In our implementation, we do not need a shared memory system or special hardware for parallel computing. The program uses only the remote shell command set, which is commonly supported in UNIX machines. We simply connected the processes through TCP/IP sockets in a 100BaseT ether network.

5 Related Works

Our method is deeply related to the universal data compression theory. Most file compression programs (e.g., "compress," "zip," etc.) are based on *Ziv-Lempel compression* [27], which was presented twenty years ago. This algorithm stores recent data in an internal table, and if the same substring appears more than once, outputs only the address of the substring stored in the table, instead of printing the substring. There are a lot of variations [1] of this method for the partitioning of substrings and the implementation of dictionaries. There are many intensive theoretical works in this field.

Our streaming BDD manipulation can also be regarded as a kind of data compression method based on BDD reduction rules. A great difference is that our BDD stream data can be processed without decompression, while most existing compression formats have to be decompressed before applying any meaningful operation. It will be interesting to discuss the connection between BDD techniques and data compression theory.

6 Conclusion

We proposed a new streaming BDD manipulation method that never causes memory overflow or swap out. This method allows us to read very large-scale BDD stream data beyond the memory limits, and the output BDD streams are concurrently produced as the input streams are read. The features of our streaming method are 1) it gives a continuous tradeoff between memory usage and stream data length, 2) a valid partial result can be obtained before completing the entire process, and 3) easily accelerated by pipelined multiprocessing.

Experimental results show that our new method is especially useful for the cases where conventional BDD packages are ineffective. For example, we succeeded in finding a number of solutions to a SAT problem using a commodity PC with 64 MB memory, where as the conventional BDD manipulator would have required a 100GB memory to solve the same problem.

BDD manipulation has been considered to be very memory-consuming procedure; and, it will be hard to get 10GB or 100GB monolithic memory blocks in the near future. The streaming method enables us to utilize disk storage and networked resources as well. The method leads to a new approach to BDD manipulation.

Currently, our streaming method has the following limitations.

- Variable order cannot be changed dynamically.
- Quantification operation (e.g., f_x & $f_{x'}$) cannot be performed in a simple pipelined process.

On the first point, dynamic variable ordering is sometimes very powerful as a way to reduce BDD size. Unfortunately, our streaming method cannot change the parsing order of Boolean space during the process. It appears to be possible to apply variable reordering to a sample BDD in memory, and then retry the streaming operations for the whole function from the beginning.

On the second point, the quantification operation requires folding of a BDD stream. This operation is hard for streaming computation to perform without unlimited random access memories. In other words, our streaming method cannot be directly applied to the symbolic model checking [5] which requires quantification operations. However, the SAT-based design verification/validation method has become a hot topic in recent years [21], [2], [16], [25], and there will be many applications that do not need quantification.

Last, another interesting future work is to consider streaming manipulation for some extended BDDs, such as MTBDDs [6], EVBDDs [9], etc.

Acknowledgments

The author would like to thank Mr. Shinya Ishihara at NTT Labs. for his helpful discussions on the pipelined multiprocessing in a PC cluster.

References

[1] T. Bell, I.H. Witten, and J.G. Cleary, "Modeling for Text Compression," *ACM Computing Surveys*, vol. 21, no. 4, pp. 557-591, Dec. 1989.
[2] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking Using SAT Procedures Instead of BDDs," *Proc. 36th ACM/IEEE Design Automation Conf. (DAC '99)*, pp. 317-320, June 1999.
[3] K.S. Brace, R.L. Rudell, and R.E. Bryant, "Efficient Implementation of a BDD Package," *Proc. 27th ACM/IEEE Design Automation Conf. (DAC '90)*, pp. 40-45, June 1990.
[4] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677-691, Aug. 1986.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *Proc. 27th ACM/IEEE Design Automation Conf. (DAC '90),* pp. 46-51, June 1990.

[6] E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang"-Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping," *Proc. 30th ACM/IEEE Design Automation Conf. (DAC '93),* pp. 54-60, June 1993.

[7] G. Janssen, "Design of a Pointerless BDD Package," *Note of Int'l Workshop Logic and Synthesis (IWLS-2001),* May 2001.

[8] S. Kimura and E.M. Clarke, "A Parallel Algorithm for Constructing Binary Decision Diagrams," *Proc. IEEE/ACM Int'l Conf. Computer Design (ICCD-90),* pp. 220-223, Sept. 1990.

[9] Y.-T. Lai, M. Pedram, and S.B. Vrudhula, "FGILP: An Integer Linear Program Solver Based on Function Graphs," *Proc. IEEE/ ACM Int'l Conf. Computer-Aided Design (ICCAD-93),* pp. 685-689, Nov. 1993.

[10] M. Löbbing and I. Wegener, "The Number of Knight's Tour Equals $33,439,123,484,294$—Counting with Binary Decision Diagrams," *The Electric J. Combinatorics,* vol. 3, no. R5, 1996.

[11] D.E. Long, "The Design of a Cache-Friendly BDD Library," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD-98),* pp. 639-645, Nov. 1998.

[12] J.C. Madre and O. Coudert, "A Logically Complete Reasoning Maintenance System Based on a Logical Constraint Solver," *Proc. Int'l Joint Conf. Artificial Intelligence (IJCAI),* pp. 294-299, 1991.

[13] K. Milvang-Jensen and A.J. Hu, "BDDNOW: A Parallel BDD Package," *Formal Method in Computer-Aided Design (Proc. FMCAD-98),* pp. 501-507, June 1998.

[14] S. Minato, "BEM-II: An Arithmetic Boolean Expression Manipulator Using BDDs," *IEICE Trans. Fundamentals,* vol. E76-A, no. 10, pp. 1721-1729, Oct. 1993.

[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. (DAC '90),* pp. 52-57, June 1990.

[16] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and A. Malik, "Chaff: Engineering an Efficient SAT Solver," *Proc. 38th ACM/ IEEE Design Automation Conf. (DAC-2001),* pp. 530-535, June 2001.

[17] H. Ochi, Y. Kouichi, and S. Yajima, "Breadth-First Manipulation of Very Large Binary-Decision Diagrams," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD-93),* pp. 48-55, Nov. 1993.

[18] H.G. Okuno, "Reducing Combinatorial Explosions in Solving Search-Type Combinatorial Problems with Binary Decision Diagram," *Trans. of Information Processing Soc. Japan (IPSJ), (in Japanese),* vol. 35, no. 5, pp. 739-753, May 1994.

[19] R.K. Ranjan, J.V. Sanghavi, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Binary Decision Diagrams on Network of Workstations," *Proc. IEEE/ACM Int'l Conf. Computer Design (ICCD-96),* Oct. 1996.

[20] B. Selman, H. Levesque, and D. Mitchell, "A New Method for Solving Hard Satisfiability Problems," *Proc. Am. Assoc. Artificial Intelligence, (AAAI '92),* pp. 440-446, July 1992.

[21] J.P.M. Silva and K.A. Sakallah, "GRASP—A New Search Algorithm for Satisfiability," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD-96),* pp. 220-227, Nov. 1996.

[22] F. Somenzi, "Efficient Manipulation of Decision Diagrams," *Int'l J. Software Tools for Technology Transfer (STTT),* vol. 3, no.2, pp. 171-181, May 2001.

[23] F. Somenzi et al. "CUDD: CU Decision Diagram Package," Public Software, http://vlsi.colorad.edu/fabio/CUDD,YEAR???

[24] T. Stornetta and F. Brewer, "Implementation of an Efficient Paralell BDD Package," *Proc. 33th ACM/IEEE Design Automation Conf. (DAC '96),* June 1996.

[25] M.N. Velev and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessor," *Proc. 38th ACM/IEEE Design Automation Conf. (DAC-2001),* pp. 226-231, June 2001.

[26] B. Yang, Y.-A. Chen, R.E. Bryant, and D.R. O'Hallaron, "Space- and Time-efficient BDD Construction via Working Set Control," *Proc. Asia and South Pacific Design Autonation Conf. (ASPDAC-98),* pp. 423-432, Feb. 1998.

[27] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Inform. Theory,* vol. 23, no. 3, pp. 337-343, Mar. 1977.

Shin-ichi Minato received the BE degree, the ME degree, and the DE degree in information science from Kyoto University in 1988, 1990, and 1995, respectively. He has been working for Nippon Telegraph and Telephone (NTT) since 1990. He was a visiting scholar at Stanford University in 1997. He served as a lecturer for Keio University from 1999 to 2001. Currently, he is an associate manager in NTT Network Innovation Laboratories. His research interests include data structures and algorithms for digital system design and verification. He published "Binary Decision Diagrams and Applications for VLSI CAD," (Kluwer, 1995). He has served as a program committee member for international conferences such as Design Automation Conference (DAC), International Conference on Computer Aided Design (ICCAD), Design, Automation and Test in Europe (DATE), etc. He is the Publicity Chair of Asia South Pacific Design Automation Design Conference (ASPDAC-2003). He is a member of the IEEE, IEICE, and IPSJ.

▷ **For more information on this or and computing topic, please visit our Digital Library at** http://computer.org/publications/dilb.