Streaming BDD Manipulation for Large-Scale Combinatorial Problems

Shin-ichi Minato and Shinya Ishihara
NTT Network Innovation Laboratories
1-1, Hikarinooka, Yokosuka-shi, 239-0847 Japan.
{minato,shinya}@exa.onlab.ntt.co.jp

Abstract

We propose a new BDD manipulation method that never causes memory overflow or swap out. In our method, BDD data are accessed through the I/O stream ports. We can read unlimited length of BDD data streams using a limited size of the memory, and the result of BDD data streams are concurrently produced. Our streaming method features that (1) it gives a continuous trade-off between the memory usage and the streaming data length, (2) a valid partial result can be obtained before completing process, and (3) easily accelerated by pipelined multi-processing.

Experimental result shows 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 a 64 MB memory, where the conventional method will require a 100 GB memory to compute it.

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. Our method will lead a new style of BDD applications.

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 efficiency in terms of time and space. A number of BDD packages (e.g. [3, 10, 14]) have been implemented and successfully applied to many real-life problems.

In conventional BDD packages, BDD data are constructed in the main memory. As repeating logic operations, the number of BDD nodes grows and grows, and sometimes abortion (or terrible performance down) occurs due to memory overflow. In general, we cannot know the peak BDD size beforehand, so we always have to be afraid of memory overflow. This is a common draw back of BDD-based applications.

The cause of memory overflow is that the BDD manipulation is based on the hash table technique to keep the uniqueness of each BDD node. The hash table works under the benefit of the random access memory, and thus, the performance falls down impractically when the memory capacity is insufficient.

In this paper, we propose a new BDD manipulation method for processing unlimited BDD nodes with a limited size of hash table. It never causes memory overflow or swap out. BDD data are accessed through the I/O stream ports. We use the main memory only for temporary working space, while conventional method constructs the whole BDD data in the memory.

Some of existing BDD packages (e.g.[14]) also have a function to save internal BDD data to a sequential file on the hard disk, however, they cannot load the BDD file beyond the main memory capacity. In our new method, we can read unlimited length of BDD data streams without memory overflow, and the result of BDD data streams are concurrently produced. Our streaming method features that (1) gives a continuous trade-off between the memory usage and the streaming data length, (2) a valid partial result can be obtained before completing process, and (3) easily accelerated by pipelined multi-processing.

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

In general, BDDs are constructed by a sequence of logic operations, starting from trivial, single-node BDDs. The binary operation algorithm[4] to generate a BDD h for the 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 the variable v and two sub-graphs generated by sub-operations $(f_{(v=0)} \circ g_{(v=0)})$ and $(f_{(v=1)} \circ g_{(v=1)})$. Repeating this expansion recursively for all the input variables, eventually trivial operations appear (e.g. $f \cdot 0 = 0$, $f \oplus f = 0$, etc.), and the results are obtained. In this recursive procedure, a number of equivalent sub-operations may appear. To avoid those redundant operations, the following two techniques are used.

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

Using these techniques, the logic operation can be carried out in a time almost linear to the number of BDD nodes.

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 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. As repeating logic operations, the BDDs grow in the memory, and sometimes fail to compute (or terrible performance down) due to memory overflow. BDD manipulation is very efficient if the memory size is sufficient, but otherwise not so.

A number of efforts have been devoted to handle large-scale BDDs beyond the memory limitation. *Breadth-first algorithm*[11] is one of the solutions to this problem. This algorithm slices the BDD nodes for each input variables, and manipulate them slice-by-slice. It reduces random accesses to the hard disk. In addition, there is a hybrid method[16] of breadth-first and the depth-first manners to improve performance. However, the breadth-first algorithm has a limitation that at least one slice of the BDDs must be stored in the same hash table to keep the uniqueness. If the "width of BDD" is too large, the memory overflow problem still remains.

There are some other works to distribute the BDD data to the networked parallel machines[6, 15, 8]. In these methods, we can handle the large-scale BDDs beyond the memory limitation of a single machine; however, they still need the total memory capacity to store all the BDD nodes.

Consequently, the existing BDD packages commonly have a limit of BDD nodes according to the memory capacity, and they cannot avoid abortion due to memory overflow.

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, suppose the bit-stream data of the truth tables for Boolean functions, as shown in Fig. 1. In this model, we can compute a logic operation bit by bit serially using no internal memory. However, the truth table representation always requires an exponential data length for an n-input function.

We then consider the streaming BDD data model, as shown in Fig. 2. The serial operation of a truth table means a scanning of Boolean space in a fixed order. This scanning corresponds to a depth-first traversal of a BDD. If we serialize the BDD data into a stream file with a depth-first traversal, we can compute a logic operation using no internal memory, as well as the truth table computation.

Figure 3 illustrates the way of serialization. If we traverse a shared sub graph every times repeatedly, the data length also become exponential as well as the truth table. However, such a duplicated traversal can be avoided in the following way: if we find a node N_k already have visited, the following traversal can be canceled just by

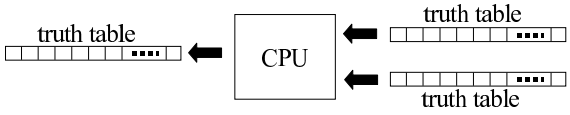

Figure 1: Streaming truth-table computation.

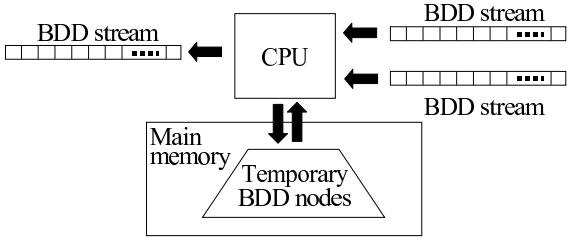

Figure 2: Streaming BDD computation.

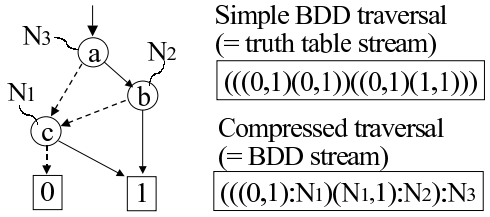

Figure 3: Serialization of a BDD.

saying "Refer to N_k" in the streaming data. To implement this idea, we need a hash table to identify all visiting nodes. In addition, if the same node appears successively twice in the traversal, we can suppress the second appearance. This is the similar idea to delete a BDD node with the same destination of 0- and 1-edge.

If the hash table size is sufficient (i.e. all the nodes can be identified), the BDD data streams are just a serialization of BDDs. The important difference is seen in the case of memory shortage. If a part of BDD nodes are missing from the hash table because of the memory limitation, we sometimes fail to know the repeated visit of a node, and a duplicated node may appear in the streaming data. This means a drop of data compression rate. More memory shortage will cause more falling down of the compression rate. Notice that, even if we have no memory for the hash table, the BDD data streams can be computed robustly as well as the truth tables. This is a great difference from the conventional BDD packages.

3.2 Data Format

Here we describe the streaming data format in our implementation. First, we specify the hash table size at the top. A BDD data stream must start with an integer `MaxID` to specify the table size, and the BDD manipulator knows the table size to see it. Each BDD node is identified by an integer ID from `1` to `MaxID`. The special ID `0` represents the 0-terminal nodes. As we use *comple-*

```
Stream ::= MaxID Inv Node
Inv ::= '~' | /* empty */
Node ::= SavedNode | TempNode
SavedNode ::= '0' | ID
        | '(' SavedNode ')'
        | '(' SavedNode Inv SavedNode '):' ID
TempNode ::= '(' Node Inv Node ')'
MaxID ::= [1-9][0-9]*
ID ::= [1-9][0-9]*
```

Figure 4: Syntax of BDD data format.

0	0
1	~0
a	(0~0):1
b	((0~0):1)
$\overline{c}$	~(((0~0):1))
$ab + \overline{c}$	~(((0~0):1)(1 0):2):3
$a \oplus b \oplus c$	(((0~0):1~1):2~2):3.
$ab + \overline{a}c$	(((0~0):1)(0~0):2):3.
$ab + bc + ac$	((0(0~0):1):2(1~0):3):4.

Figure 5: Simple examples of BDD data streams.

ment edges[10], the 1-terminal is expressed as ~0.

Figure 4 shows the syntax of the data format in a BNF-like description[1]. In this format, we describe a node by a pair of parentheses enclosing the two child nodes (0-child 1-child). The nested parentheses represent the structure of the graph. '$\sim$' is the inverter to the following node. ':' defines an ID to the latest node and registers it to the node table. `SavedNode` expresses a node already stored in the table, and `TempNode` is a temporary node to be lost immediately. A `SavedNode` cannot have a `TempNode` in its child. An `ID` must be referred 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 parentheses indicates the corresponding variable[2]. Figure 5 shows some simple examples of BDD data streams.

If the original BDD nodes are no more than `MaxID`, the streaming BDD data uniquely represent Boolean functions under a fixed variable ordering. If the table size is insufficient, the streaming data may become different representations for the same BDD. For example, Fig. 6 show the streaming BDD data for the same function in different `MaxID`s: 30, 20, and 10. The original BDD requires 24 nodes, so the table overflow occurs when `MaxID` = 20 or 10. In such cases, our format allows the recycle use of a "orphan" node ID, which is not referred from other nodes. For example, when `MaxID` = 10, the node '1' to the '10' are stored in the table normally, but there is no space to save the 11th node. We then erase the orphan node '9' and recycle the ID for the latest node. Consequently, the node '6' newly becomes orphaned, so it can be recycled on the next time. If there are multiple

[1]Here we show a plain text format for easy debugging. A binary format will be more compact.

[2]Our implementation employs run-length compaction for successive parentheses: e.g. '((((((' into '(*6'.

```
30
(((((((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.

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).

10
(((((((0(0(0~0):1):2):3(2(1~0):4):5):6(5(4~0):7)
:8):9(8(7~0):10):9)(9(10~(0 3):6):9)))((8 10):9
(10~6):9)(9~(6(3 5):8):9)))(((((5 7):10(7~0):9)
:6(9~(0 3):8):6)(6~(8(3 5):10):6))(((9~8):6~(8
10):6)~(6(10(5 7):9):6)))).
```

Figure 6: Streaming data for "9sym".

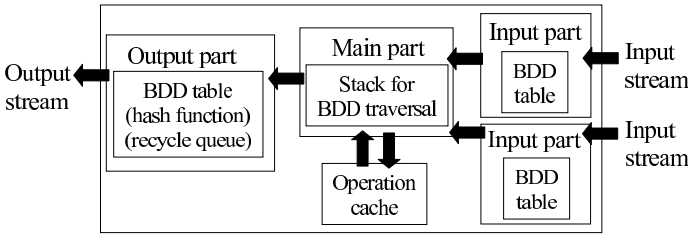

Figure 7: Internal structure of the program.

candidates to be recycled, we choose one waiting for the longest time. In this way, we can use the limited memory space efficiently.

3.3 Logic Operation

Figure 7 shows the internal structure of our implementation for binary logic operation. It has two BDD tables for the input parts and one BDD table 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 table size of the output part can be set up by hand (specified in the command option).

The output part has a hash-based unique table and a recycle queue to control memory usage. The input part does not have such devices and simply reconstruct BDD data sent from the upper-stream operation. At first, we start parsing of the input data and store the BDD structure into the internal table. When a stored node ID reappears in the input data, we suspend parsing and switch the traversal to the internal table. After traversal of the sub graph in the table, we resume parsing of the input data.

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

In the output part, we must consider the following

case: when creating a new node, the data may be inconsistent since a child node might have been recycled. To detect such a case, we attach a time-stamp to each node to check the recycled use. If we detect inconsistency of a child node, we produce a `TempNode` instead of `SavedNode`. When the BDD table size is much more insufficient, the node recycling occurs more frequently, more `TempNode`s are produced, and the output data grows longer. This method gives a continuous trade-off curve between the memory usage and the streaming data compression rate.

Here we have discussed on the binary logic operations, but it is easily extended to the ternary (3-input) operations by adding one more input part. The use of a ternary operation will reduce computation time comparing to twice of binary operations, although we need an additional memory space for the extra input part. The 4-input (and more) operations are also possible in the same way.

4 Experimental Results

We implemented a logic operation program to manipulate BDD data streams on the UNIX environment. The program, named `BDDstrm`, is written in 2,000 lines of C code. At first we write some trivial BDD stream files, and we then repeatedly execute `BDDstrm` to construct the objective BDD streams. In the UNIX environment, we can conveniently use the pipe connection of the two or more processes in a command line.

`BDDstrm` has an option to limit the output data length. The program automatically aborts at the specified limit to prevent hard disk overflow. In this case, or whenever we quit the process halfway, the incomplete output data represents a valid result for partial Boolean space. We can continue to apply the next operation to the incomplete output data stream. This is another feature of our streaming manipulation method.

In addition, we implemented a program to save or load our BDD data streams from/to a conventional BDD manipulator. This enables us to link our new method to the existing BDD-based programs.

4.1 Basic Performances

Here we summarize the basic performances 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 in a 64MB memory.)
- **Streaming data length:**
  5 to 15 Byte/node.
  (about a million BDD nodes in 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.

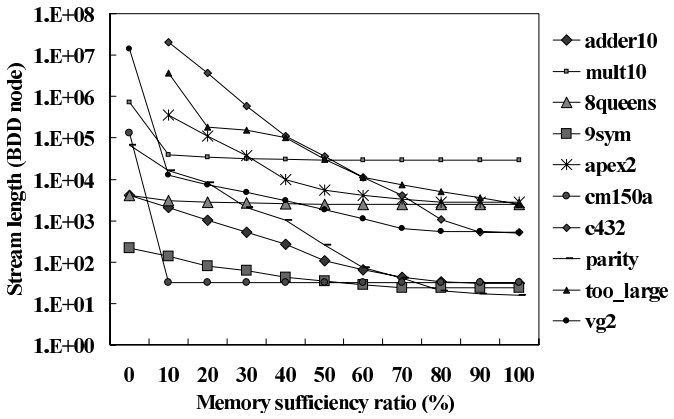

Figure 8: Trade-off between memory vs. data length.

4.2 Trade-off: Memory vs. Data Length

As discussed in Section 3, the output BDD data grows longer when the hash table size is insufficient. To show the trade-off curve of the data length for the table size shortage, we conducted the following experiment. First we provide a sufficient size of the hash table and count the number of BDD nodes written in the output data. We then gradually decrease the hash table size to observe how the output data grows.

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

In this experiment, we can see different trade-off curves depending on the functions. For example, "mult10", "8queens", and "cm150a" are not so sensitive to the memory shortage up to only 10% or less. This means that we can efficiently handle more than ten times larger BDDs beyond the memory capacity. On the other hand, "parity", "c432", and "too_large" are very sensitive. In general, more shared BDDs are more sensitive to the memory shortage.

Here we emphasize that the conventional BDD manipulation is anyway faced with memory overflow problem even if the sufficiency ratio was 99%. Although our method has some overhead in terms of data length or computation time, it enables us to avoid the unpredictable memory overflow.

4.3 Solving SAT Problems

Many problems in LSI CAD 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, and 0-1 linear programming are the popular examples. SAT-based design verification/validation is also a new topic[13, 2] in recent years.

Table 1: Experimental results for solving N-Queens Problem.

N	#Var	Prev.	Our method(No File Limit)				(Lim:10MB)		(Lim: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

(Ultra SPARC 30, 128MB Mem, 2.5GB free HD, SunOS 5.6)

Table 2: Pipelined multi-processing.

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)

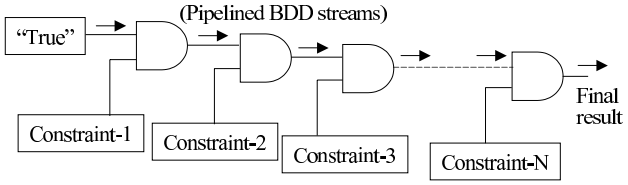

Figure 9: Solving a SAT problem.

There are several works (e.g. [9, 12]) to solve SAT problems using BDD manipulation. In the method, we 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 to satisfy all the constraints. Unfortunately, we are sometimes faced with memory overflow for large-scale problems.

We implemented a SAT-problem solver based on the streaming BDD manipulation. As shown in Fig.9, we prepare a BDD stream file for each constraint, and compute the conjunction by a cascade of streaming BDD processors. In this system, an intermediate BDD stream represents the "candidates" of solutions satisfying the constraints have 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 of processors may produce duplicated nodes when the memories are not sufficient, however, those redundant nodes can be eliminated in the lower stream if the final result of BDD is not very complicated. For example, if the problem is not satisfiable, the simple result "0" is produced from the final processor after the whole data have been processed. In other words, when the first BDD node appears at the final output, immediately we know the satisfiability. For a complicated problem, the intermediate streams may grow unlimitedly long, and we cannot know when it will be completed. If we quit the process halfway, the incomplete output data contains a partial set of solutions. This is a great difference from the conventional BDD manipulation, which gives no solutions in the case of overflow.

To evaluate the effect of our streaming manipulation, we conducted experiments of solving N-Queens problems, as shown in Table 1. An N-Queens problem has N^2 places on the chessboard, so we prepared N^2 Boolean variables and N^2 constraints to specify the problem. We then compute the conjunction of all the Boolean constraints. The column "Prev." shows the CPU time 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 the solutions up to 14-Queens beyond the memory limitation. In addition, with the limited length of BDD stream files, we could generate a partial set of solutions for larger N's in a feasible computation time. This is another great benefit of our method.

If we use a single processor, we have to save a BDD streaming file into the hard disk on each logic operation. Using multiple PCs, we can deploy the logic operations to the PCs and directly connect their I/O ports. In this way, the computation can greatly be accelerated according to the number of PCs. Table 2 shows the results of pipelined multi-processing on a networked PCs. In this pipelining approach, we do not need shared memories or special devices for parallel computing. In our experiment, we only used "rsh" and "mkfifo", which are commonly supported in UNIX systems.

5 Related Works

Our method is deeply related to the universal data compression theory. Most of file compression programs (e.g. "compress", "zip", etc.) are based on *Ziv-Lempel compression*[17], presented in twenty years ago. This algorithm stores the recent data into the internal table, and if the same sub-string appears more than once, puts out only the address of the sub-string stored in the table, instead of printing the sub-string. There are a lot of variations[1] of this method on the partitioning of sub-strings and the implementation of the dictionary. There are many intensive theoretical works in this field.

Our streaming BDD manipulation can also be regarded as a kind of the data compression method based on the BDD reduction rules. A big difference is that

our BDD data streams can be processed without decompression, while usual compressed data files have to be decompressed before applying meaningful operations. It will be interesting to discuss the BDD techniques on the aspect of data compression theory.

6 Concluding Remarks

BDD manipulation has been considered as an intensively memory-consuming procedure; however, it will be hard to get a 10GB or 100GB monolithic memory block in the near future. The streaming manipulation enables us to utilize disk storage and networked resources as well. Our method will lead a new style of BDD applications.

Currently, our streaming method has the following limitations.

- Variable order cannot be changed dynamically.
- Quantification operation (e.g. AND-EXIST) cannot be executed in a simple pipelined processing.

On the first point, dynamic variable ordering is sometimes very powerful to reduce the BDD size. Unfortunately, our streaming method cannot change the parsing order of Boolean space during the process. It will be a good way to apply variable reordering for a sample BDD on the 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 is hard for streaming computation without unlimited random access memories. In other word, it is so far difficult to directly apply our streaming method to the symbolic model checking. However, SAT-based design verification/validation method is also a new topic in recent years[13, 2], and there are many other LSI CAD applications in the NP or co-NP class. Our streaming method is useful for those applications.

Lastly, it will be another interesting future work to consider streaming manipulation for some extended BDDs, such as MTBDDs[5] or EVBDDs[7].

References

[1] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):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. In *Proc. of 36th ACM/IEEE Design Automation Conference (DAC'99)*, pages 317–320, June 1999.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pages 40–45, June 1990.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

[5] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pages 54–60, June 1993.

[6] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. on IEEE/ACM International Conference on Computer Design (ICCD-90)*, pages 220–223, Sept. 1990.

[7] Y.-T. Lai, M. Pedram, and S. B. Vrudhula. FGILP: An integer linear program solver based on function graphs. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pages 685–689, Nov. 1993.

[8] K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel BDD package. In *Formal Method in Computer-Aided Design (Proc. of FMCAD-98)*, LNCS-1522, pages 501–507. Springer, June 1998.

[9] S. Minato. BEM-II:an arithmetic Boolean expression manipulator using BDDs. *IEICE Trans. Fundamentals*, E76-A(10):1721–1729, Oct. 1993.

[10] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pages 52–57, June 1990.

[11] H. Ochi, Y. Kouichi, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pages 48–55, Nov. 1993.

[12] H. G. Okuno. Reducing combinatorial explosions in solving search-type combinatorial problems with binary decision diagram. *Trans. of Information Processing Society of Japan (IPSJ), (in Japanese)*, 35(5):739–753, May 1994.

[13] J. P. M. Silva and K. A. Sakallah. GRASP—a new search algorithm for satisfiability. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-96)*, pages 220–227, Nov. 1996.

[14] F. Somenzi et al. CUDD: CU decision diagram package. Public Software. `http://vlsi.colorad.edu/~fabio/CUDD`.

[15] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *Proc. of 33th ACM/IEEE Design Automation Conference (DAC'96)*, June 1996.

[16] B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. O'Hallaron. Space- and time-efficient BDD construction via working set control. In *Proc. of Asia and South Pacific Design Automation Conference (ASPDAC-98)*, pages 423–432, Feb. 1998.

[17] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, IT-23(3):337–343, Mar. 1977.