The I/O-Complexity of Ordered Binary-Decision Diagram Manipulation

(Extended Abstract)

Lars Arge[*]

BRICS[**], Department of Computer Science, University of Aarhus, Aarhus, Denmark

Abstract. We analyze the I/O-complexity of existing Ordered Binary-Decision Diagram manipulation algorithms and develop new efficient algorithms. We show that these algorithms are optimal in all realistic I/O-systems.

1 Introduction

Ordered Binary-Decision Diagrams (OBDDs) [6, 7] are the state-of-the-art data structure for boolean function manipulation and they have been successfully used to solve problems from numerous areas like e.g. digital-systems design, verification and testing, mathematical logic, concurrent system design and artificial intelligence [7]. There exist implementations of OBDD software packages on a number of sequential and parallel machines [4, 5, 14, 15]. Even though there exist very different sized OBDD representations of the same boolean function, OBDDs in real applications tend to be very large. In [4] for example OBDDs of Gigabyte size are manipulated in order to verify logic circuit designs, and researchers in this area would like to be able to manipulate orders of magnitude larger OBDDs. In such cases the Input/Output (I/O) communication becomes the bottleneck in the computation.

Until recently most research, both theoretical and practical, have concentrated on finding small OBDD-representations of boolean functions appearing in specific problems [5, 7, 13, 16], or on finding alternative succinct representations while maintaining the efficient manipulation algorithms [10]. Very recently however, researchers have begun to consider I/O-issues arising when the OBDDs get larger than the available internal memory, and experimental results show that very large speedups can be achieved with algorithms that try to minimize the access to external storage as much as possible [4, 15]. These speedups can be achieved because of the extremely large access time of external storage medias, such as disks, compared to the access time of internal memory. In the coming years we will be able to solve bigger and bigger problems due to the development of machines with larger and faster internal memory and due to increasing CPU

[*] This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II). Part of the work was done while a Visiting Scholar at Duke University. Email: `large@daimi.aau.dk`.

[**] Acronym for Basic Research in Computer Science, a Center of the Danish National Research Foundation.

speed. This will however just increase the significance of the I/O-bottleneck since the development of disk technology lack behind developments in CPU technology. At present, technological advances are increasing CPU speed at an annual rate of 40-60% while disk transfer rates are only increasing by 7-10% annually [17].

1.1 I/O-model and Previous Results

In this paper we will be working in the parallel I/O-model introduced by Aggarwal and Vitter [1] which models the I/O-system of many existing workstations. The model has the following parameters:

$$\begin{aligned} N &= \#\text{ of elements in the problem instance} \\ M &= \#\text{ of elements that can fit into main memory} \\ B &= \#\text{ of elements per disk block} \\ D &= \#\text{ of disks} \end{aligned}$$

An I/O-operation in the model is the process of simultaneously reading or writing a block of data to or from each of the D disks [20]. The total amount of data transferred in one I/O is thus DB elements. The I/O-complexity of an algorithm is simply the number of I/Os it perform. Internal computation is free, and we always assume that the N elements initially are stored in the first $N/(DB)$ blocks on each disk. Typical values for workstations and file servers in production today are on the order of $10^6 \leq M \leq 10^8$, $B \simeq 10^3$ and $D \simeq 10^1$.

Early work on external-memory algorithms concentrated on sorting and permutation-related problems [1, 20]. More recently researchers have designed external memory algorithms for a number of problems in different areas. Most notably I/O-efficient algorithms have been developed for a large number of computational geometry [3, 12] and graph problems [9]. Also worth noticing in this context is [11] that addresses the problem of storing graphs in a paging-environment, but not the problem of performing computation on them, and [2] where a number of external (batched) dynamic data structures are developed. Finally, it is demonstrated in [8, 19] that the results obtained in the mentioned papers are not only of theoretical but also of great practical interest.

Some notes should be made about the typical bounds in the model. While $N/(DB)$ is the number of I/Os needed to read all the input, $\Theta(\frac{N}{DB}\log_{M/B}\frac{N}{B}) = \Theta(sort(N))$ is the number of I/Os needed to sort N elements [1]. Furthermore, the number of I/Os needed to rearrange N elements according to a given permutation is $\Theta(\min\{N/D, sort(N)\}) = \Theta(perm(N))$. Taking a closer look at these bounds for typical values of B, M, and D reveals that $\log_{M/B}\frac{N}{B}$ is less than 3 or 4 for all realistic values of N. This means that the sorting term in the permutation bound in all realistic cases will be smaller than N/D, such that $perm(N) = sort(N)$. The term in the bounds that really makes the difference in practice is the DB-term in the denominator. Normally we will be willing to accept an $\log_{M/B}\frac{N}{B}$ term in order to go from a bound like $O(N)$, where we generate a page-fault on every memory access, to a bound with a DB term in the denominator.

1.2 OBDDs and Previous Results

A branching program is a directed acyclic graph with one source (the root), where the sinks are labeled by boolean constants. The inner vertices are labeled by boolean variables and have two outgoing edges labeled with 0 and 1. If a vertex is labeled with x_i we say that it has index i. The evaluation of an input $a = (a_1, \ldots, a_n)$ starts at the root. At a vertex labeled x_i the outgoing edge with label a_i is chosen. The label of the sink reached this way equals $f(a)$ for the boolean function f represented by the branching program. An OBDD is a branching program for which an ordering of the variables in the vertices is fixed. For simplicity we assume that this ordering is the natural one, $x_1, \ldots, x_n$. If a vertex with label x_j is a successor of a vertex with label x_i, the condition $j > i$ has to be fulfilled. Note that an OBDD representing a boolean function of n variables can be of size 2^n, and that different variable orderings lead to representations of different size. There exist several algorithms (using heuristics) for choosing a variable-ordering that minimize the OBDD-representation of a given function [13, 16].

In [6] Bryant proved that for a given variable ordering and a given boolean function there is (up to isomorphism) exactly one OBDD — called the reduced OBDD — of minimal size. He also showed that the there are two fundamental operations on OBDDs — the *reduce* and the *apply* operation. The apply operation takes the OBDD-representation of two functions and computes the representation of the function formed by combining them with a binary operator. Finally, Bryant proved that using the following two reduction rules on a OBDD with at most one 0-sink and one 1-sink, until none of them are applicable anymore, yields the reduced OBDD: 1) If the two outgoing edges of vertex v lead to the same vertex w, then eliminate vertex v by letting all edges leading to v lead directly to w. 2) If two vertices v and w labeled with the same variable have the same 1-successor and the same 0-successor, then merge v and w into one vertex. Existing apply and reduce algorithms run in $O(|G|)$ and $O(|G_1| \cdot |G_2|)$ time, respectively [4, 6, 7, 14, 15, 18]. Here $|G|$ denote the size (the number of vertices) of the OBDD G.

Even though the I/O-system (the size of the internal memory) seems to be the primary limitation on the size of the OBDD problems one is able to solve practically today [4, 5, 6, 7, 15], it was only very recently that OBDD manipulation algorithms especially designed to minimize I/O was developed. In [4] and [15] it is realized that the traditional algorithms working in a depth-first or breadth-first manner on the involved OBDDs does not perform well when the OBDDs are too large to fit in internal memory, and new level-wise algorithms are developed.[3] The general idea in these algorithms is to store the OBDDs in a level-blocked manner,[4] and then try to access the vertices in a pattern that is as

[3] When we refer to depth-first, breadth-first and level-wise algorithms we refer to the way the apply algorithm traverse the OBDDs. All known reduce algorithms work in a level-wise manner.

[4] When we say that a OBDD is e.g. level-blocked we mean that the OBDD is stored in such a way that a given block only contain vertices from one level.

level-wise as possible. Previous algorithms did not explicitly block the OBDDs. In [4, 15] speed-ups of several hundreds compared to the "traditional" algorithms are reported using this idea.

In the rest of this paper we assume that an OBDD is stored as a number of vertices and that the edges are stored implicitly in these. We also assume that each vertex knows the index (level) of its two sons. This means that the fundamental unit is a vertex (e.g., an integer — we call it the *id* of the vertex) with an index, and a pointer and an index for each of the two sons. We assume that a pointer to a son is just the id of the son. The vertices also contain a few other fields used by the apply and reduce algorithms. Finally, we assume that a reduce operation is done as part of the apply operation after the actual apply operation is performed.

1.3 Our Results

In this paper we analyze the I/O-performance of the existing OBDD manipulation algorithms and develop new I/O-efficient algorithms. We show that all existing reduce and apply algorithms have a bad worst case performance, that is, they use $\Omega(|G|)$ and $\Omega(|G_1| \cdot |G_2|)$ I/Os in the worst case, respectively. We show that this is even true if "good" blockings are assumed — that is, if e.g. a depth-first blocking is assumed when analyzing the depth-first algorithm — and also true for the algorithms especially designed with I/O in mind. The main contribution however, is that we show that under some natural restrictions (fulfilled by all existing algorithms) $\Omega(perm(|G|))$ is a lower-bound on the number of I/Os needed to reduce an OBDD of size $|G|$. We show that this is the case even if we assume that the OBDD being reduced is depth-first, breadth-first or level-blocked, and even if we assume another intuitively good/optimal blocking which we will define later. Finally, we develop a new reduction algorithm that use $O(sort(|G|))$ I/Os, and a new apply algorithm that use $O(sort(|G_1| \cdot |G_2|))$ I/O-operations. As discussed in section 1.1 this is asymptotically optimal for all realistic I/O-systems. We believe that the developed algorithms are of practical interest due to relatively small constants in the asymptotic bounds.

2 The Reduce Operation

All the reduction algorithms [4, 6, 7, 14, 15] reported in the literature basically work in the same way. They all process the vertices level-wise from the sinks up to the root and assign a (new) unique integer label to each unique sub-OBDD root. Processing a level, under the assumption that all higher levels have already been processed, is done by looking at the new labels of the sons of the vertices on the level in question, checking if the reduction rules can be used, and assigning new labels to the vertices. While checking if reduction rule one can be used is easy, the algorithms either sort the vertices according to the labels of the sons, or maintain a (hash) table with an entry for each unique vertex generated so far, in order to check if reduction rule two can be used.

It is fairly easy to realize that all the existing algorithms perform badly in an I/O-environment, even assuming that the OBDDs are initially blocked in some "good" way. There are several reasons for this — for example the way some of the algorithms traverse the OBDD, or the "random" lookups in the table of unique vertices. The main reason however, is the visiting of the sons during the processing of a level of vertices. As there is no "nice" pattern in the way the sons are visited (mainly because a vertex can have large fan-in), the algorithms can in the worst case be forced to do an I/O each time a son is visited. The algorithms specifically developed with I/O in mind try to avoid some off all these page faults by visiting the sons in level order. This follows the general philosophy mentioned earlier that vertices should be visited level-wise. But still it is not hard to realize that also these algorithms could be forced to do an I/O every time a son is accessed, because there is no correlation between the order in which the sons on a given level are visited and the blocks they are stored in. In the full version of this paper a more complete analysis of the existing reduce algorithms is done, and it is shown that they all cause $\Omega(|G|)$ page-faults in the worst case when reducing an OBDD of size $|G|$.

2.1 I/O-Lower Bound on the Reduce Operation

In this section we will sketch the proof of the I/O-lower bound on the reduce operation. We prove the lower bound under the assumption that the reduce algorithm works like the known algorithms, that is, it works by assigning new labels to vertices and checking if one of the reduction rules can be used on vertices whose sons have already been assigned new labels. The precise assumption is that the new label is assigned to the original vertex, and that the sons are loaded into internal memory (if they are not there already) in order to obtain their new labels.

In [9] the *proximate neighbors problem* is defined as follows: We are given N elements in external memory, each with a key that is a positive integer $k \leq N/2$. For each possible value of k, exactly two elements have that key-value. The problem is to permute the elements such that elements with identical key-value are in the same block. We define a variant of the proximate neighbors problem called the *split proximate neighbors (SPN) problem.* This problem is defined similar to the proximate neighbors problem, except that we require that the keys of the first $N/2$ elements in external memory (and consequently also the last $N/2$ elements) are distinct. Following the I/O-lower bound proof on the proximate neighbors problem in [9] we can prove the following:

Lemma 1. *Solving the SPN problem require $\Omega(perm(N))$ I/Os in the worst case.*

Using Lemma 1 we can now prove the lower bound by reducing the SPN problem to the reduction problem. Given an SPN problem we create and block an unreduced OBDD as sketched in figure 1 and 2a). Figure 1 indicates how the SPN problem $8, 1, 2, 4, \cdots, 7, 6, 3, 5, 3, 8, 4, 7, \cdots, 1, 5, 3, 6$ is encoded in the highest levels of the OBDD. On top of the *base-blocks*, as we will call the blocks storing the highest levels of the OBDD, we build a complete tree. Figure 2a)

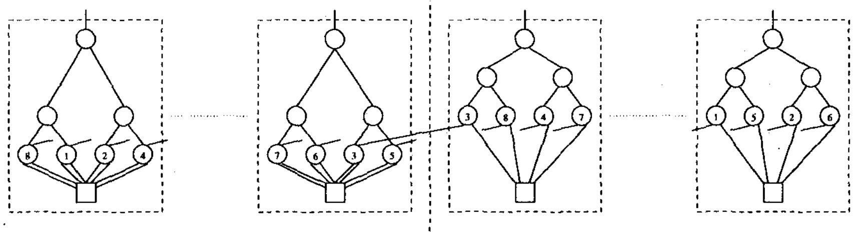

Fig. 1. Highest levels of the OBDD for a SPN problem ($B = 8$).

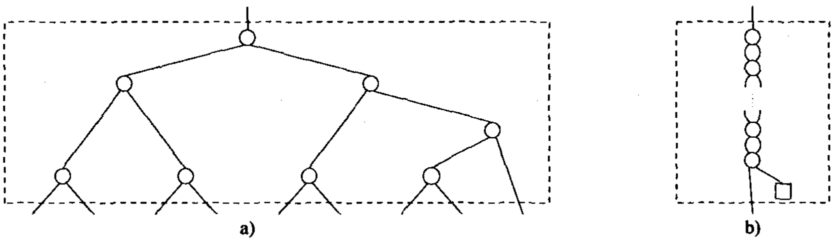

Fig. 2. a) One block of top blocking. b) Block used in proof of Lemma 4.

shows one block in the blocking of this tree. The blocking indicated on the figures is a breadth-first blocking of the OBDD. It is relatively easy to realize that if we construct the OBDD such that the id of every vertex containing an element from the first half of the SPN problem is the key of the element it holds, then the OBDD can be constructed and blocked in $O(|G|/(DB))$ I/Os. Now, whatever reduce algorithm we run on the produced OBDD, it will at some point have to hold both vertices with the same SPN problem key in main memory. This is due to our assumption about the reduce algorithm — the vertices containing elements from the last half of the SPN problem will at some point have to check the new label of their sons. This means that we can augment any reduce algorithm to solve the SPN problem.

In the full paper we give all the details in the above construction and show how we can also produce a level blocked or even depth-first blocked OBDD from an SPN problem in $O(|G|/(DB))$ I/Os. We also discuss precisely in which model of computation the construction works. As an algorithm for the SPN problem constructed in the above way use $O(|G|/(DB)) + O(\mathit{reduce}(|G|))$ I/Os, the following lower bound follows from Lemma 1.

Lemma 2. *Reducing an OBDD with $|G|$ vertices requires $\Omega(\mathit{perm}(|G|))$ I/Os in the worst case — even if the OBDD is depth-first, breadth-first, or level blocked.*

Even though Lemma 2 provides us with a lower bound on reducing an OBDD blocked in one of the natural ways for the different apply algorithms (and even in the way the OBDDs are actually blocked in the algorithms especially designed with I/O in mind), it does not mean that we could not be lucky and be able to reduce an OBDD in less I/Os, presuming that we had it blocked in some other clever way. Intuitively the best blocking strategy, under the assumptions on the reduce algorithm we have made, would be to minimize the number of neighbor vertices in different blocks (vertices connected by an edge in different blocks). We will call such a blocking a *minimal-pair blocking.* But as we will prove next, a slightly modified version of the breadth-first blocking we just considered — where a layer of the blocks pictured in figure 2b) is inserted between the base-blocks and the blocked tree — is in fact such a minimal-pair blocking for the OBDD in question. This means that the lower bound also holds for minimal-pair blockings.

Intuitively the blocking in figure 1 and 2 is a minimal-pair blocking because all blocks except for the base-blocks only have one in-edge, and because the vertices from the base-blocks cannot be blocked in a better way than they are, that is, the vertices contained in one base-block must account for at least $B/2+1$ in or out-edges — what we will call *pair-breaking* edges. We will sketch the proof of the last statement. Call the vertices containing the first $N/2$ elements of the split proximate neighbors problem for a-vertices, and the vertices containing the last $N/2$ elements for b-vertices. All these vertices are the "special" vertices because they have edges to each other and to the multi fan-in sink-vertices. Now call a sink-vertex that is connected with an a or b-vertex a c-vertex, and consider a block in an arbitrary blocking of the OBDD in figure 1 and 2.

Lemma 3. *Let K be a block containing a_1 a-vertices and their corresponding c-vertices and a_2 a-vertices without their corresponding c-vertices, together with b_1 b-vertices with their c-vertices and b_2 b-vertices without their c-vertices. Assume furthermore that a_1, a_2, b_1 and b_2 are all $\leq B/2$, and that at least one of the a_i's and one of the b_i's are non-zero. K has at least $a_1+a_2+b_1+b_2+k$ pair-breaking edges, where k is the number of a_1, a_2, b_1, b_2 that are non-zero.*

Sketch of proof: First we assume without loss of generality that all the a_1 a-vertices are from the same base-block. This is the best case as far as pair-breaking edges are concerned because we only need one c-vertex for all the a_1 a-vertices. We assume the same about the a_2 a-vertices and the b_1 and b_2 b-vertices, ending up with vertices from at most four base blocks.

We divide the proof in cases according to the value of a_1 and b_1. We illustrate the general idea in the proof with the case $a_1 \geq 1$ and $b_1 \geq 1$. In this case we need $a_1+a_2+b_1+b_2+2$ of K's capacity of B vertices to hold the a and b-vertices and the two c-vertices. It can now be realized that the number of pair-breaking edges in a block consisting of *only* these vertices is $a_1+2(B/2-a_1)+3a_2+b_1+(B/2-b_1)+2b_2+|(a_1+a_2)-(b_1+b_2)|$. Now we add vertices one by one in order to obtain the final block K. Because of the extra blocks (figure 2b) the number of pair-breaking edges can now only be reduced by one for each new vertex we add. The lemma then follows from the fact that $(a_1+2(B/2-a_1)+3a_2+b_1+(B/2-b_1)+2b_2+|(a_1+a_2)-(b_1+b_2)|)-(B-(a_1+a_2+b_1+b_2+2)) \geq a_1+a_2+b_1+b_2+k$.
□

In the full paper we show how Lemma 3 can be extended to the case where one of the variables is greater than $B/2$, and the case where all vertices in the block are of the same type, and that this leads to the following lemma:

Lemma 4. *The blocking of the OBDD showed in figure 1 and 2 is a minimal-pair blocking.*

To summarize, we have proved the following:

Theorem 5. *Reducing an OBDD with $|G|$ vertices - arbitrarily, level, depth-first, breadth-first or minimal-pair blocked - requires $\Omega(perm(|G|))$ I/Os in the worst case.*

2.2 I/O-Efficient Reduce Algorithm

Recall that the main problem with all the known apply algorithms with respect to I/O, is that when they process a level of the OBDD, they do a lot of I/Os in order to obtain the new labels of the sons of the vertices on the level in question. Our solution to this problem is simple — when a vertex is given a new label we "inform" all its immediate predecessors about it in a "lazy" way using a priority queue. Every time we give a vertex v a new label, we insert an element in the I/O-efficient priority-queue developed in [2] for each of the immediate predecessors of v. The elements in the priority-queue are ordered according to level and vertex-id of the "receiving" vertex, such that when we on a higher level want to know the new labels of sons of vertices on the level, we simply do deletemin operations on the queue until we have obtained all elements on the level in question. As the number of operations performed on the queue is linear in $|G|$, it follows from the $O((\log_{M/B} N/B)/(DB))$ amortized I/O-bound on the insert and deletemin operation proven in [2] that we overall use $O(sort(|G|))$ I/Os to manipulate the queue. In the full paper we will give all the details in the algorithm and prove the following:

Theorem 6. *An OBDD with $|G|$ vertices can be reduced in $O(sort(|G|))$ I/Os.*

As mentioned in the introduction, Theorem 5 and Theorem 6 together means that for all realistic I/O-systems we have developed an asymptotically optimal reduction algorithm.

3 The Apply Algorithm

The basic idea in all existing apply algorithms [4, 6, 7, 14, 15] is to use the recursive formula $f_1 \otimes f_2 = \overline{x_i} \cdot (f_1|_{x_i=0} \otimes f_2|_{x_i=0}) + x_i \cdot (f_1|_{x_i=1} \otimes f_2|_{x_i=1})$ to design a recursive algorithm. Here $f|_{x_i=b}$ denotes the function obtained from f when the argument x_i is replaced by the boolean constant b, and $\otimes$ is some binary operator. Using this formula Bryant [6] developed a recursive algorithm working in a depth-first manner on the involved OBDDs. In [14] and [4, 15] algorithms working in a breadth-first and in a level-wise manner are then developed. These algorithms work like Bryant's except that recursive calls (vertices who need to have their sons computed — we call them *requests*) are inserted in a global queue and in a queue for the level they should be computed on, respectively, and computed one at a time. In order to avoid generating the OBDD for a pair of sub-OBDDs more than once — which would result in exponential (in n) running time — all the algorithms use dynamic programming (maintain a table of size $|G_1| \cdot |G_2|$). Before doing a recursive call Bryant's algorithm tests if a simular call has been done before. Similarly the other algorithms test if a similar request already exists before inserting a new request in a queue.

It is relatively easy to realize that in order to hope for a good I/O-performance of the above sketched algorithms, we need to have the involved OBDDs blocked in a manner that corresponds to the algorithm that work on them — depth-first, breadth first, or level-wise. However, the algorithms still perform poorly mainly because there is no nice structure in the way the dynamic programming table is accessed — not even in the way one level of it is accessed. In the full paper we discuss this in detail and show that all the known algorithms in the worst case use $\Omega(|G_1| \cdot |G_2|)$ I/Os when trying to do an apply operation.[5]

3.1 I/O-Efficient Apply Algorithm

The main idea in our apply algorithm is to do the computation level-wise as in [4, 15], but to use a priority-queue to control the recursion. Using a priority-queue we do not need to have a queue for each level. In order to avoid the random lookups in the dynamic programming table, we do not check for duplicate requests when new requests are generated, but when they are about to be computed. This is done by ordering the priority-queue not only according to level, but secondary according to the id's of the nodes in the requests. In order to process one level of the requests we simply do deletemin operations on the priority-queue, and remove equal requests that will now appear next to each other in the order they are deleted from the queue. This way we get rid of the dynamic programming table. As in the case of the reduce algorithm we in total do a linear number of operations on the priority queue, and this and a careful analysis of the details in the algorithm leads to the following result:

Theorem 7. *The apply operation can be performed on two OBDDs G_1 and G_2 in $O(sort(|G_1| \cdot |G_2|))$ I/Os.*

Acknowledgments

The author would like to thank Allan Cheng for introducing him to OBDDs, Darren Vengroff and Jeff Vitter for inspiring discussions, and Peter Bro Miltersen for comments on earlier drafts of this paper.

[5] The natural question is of course why experiments with the level algorithms show so huge speedups compared to the traditional algorithms. The answer is partly that the traditional depth-first and breadth-first algorithms behave so poorly with respect to I/O that just considering I/O-issues, and actually try to block the OBDDs and access them in a "sequential" way, leads to large speedups. However, we believe that one major reason for the experimental success in [4] is that the OBDDs in the experiments roughly are of the size of the internal memory of the machines used. This means that one level of the OBDDs actually fits in internal memory. This again explains the good performance because the worst case behavior occurs when one level of the OBDD (or the dynamic programming table) does not fit in internal memory.

References

1. A. Aggarwal, J.S. Vitter: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM, 31 (9), 1988.
2. L. Arge: The Buffer Tree: A New Technique for Optimal I/O-Algorithms. In Proc. of 4th Workshop on Algorithms and Data Structures, 1995.
3. L. Arge, D.E. Vengroff, J.S. Vitter: External-Memory Algorithms for Processing Line Segments in Geographic Information Systems. In Proc. of 3rd Annual European Symposium on Algorithms, 1995.
4. P. Ashar, M. Cheong: Efficient Breadth-First Manipulation of Binary Decision Diagrams. In Proc. of 1994 IEEE International Conference on CAD.
5. S.K. Brace, R.L. Rudell, R.E. Bryant: Efficient Implementation of a BDD Package. In Proc. of 27'th ACM/IEEE Design Automation Conference, 1990.
6. R. Bryant: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on computers, C-35 (8), 1986.
7. R. Bryant: Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. ACM Computing Surveys, 24 (3), 1992.
8. Y.-J. Chiang: Experiments on the Practical I/O Efficiency of Geometric Algorithms: Distribution Sweep vs. Plane Sweep. In Proc. of 4th Workshop on Algorithms and Data Structures, 1995.
9. Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, J.S. Vitter: External-Memory Graph Algorithms. In Proc. of 6th ACM/SIAM Symposium on Discrete Algorithms, 1995.
10. J. Gergov, C. Meinel: Frontiers of Feasible and Probabilistic Feasible Boolean Manipulation with Branching Programs. In Proc. of 10th Symposium on Theoretical Aspects of Computer Science, LNCS 665, 1993.
11. M.T. Goodrich, M.H. Nodine, J.S. Vitter: Blocking for External Graph Searching. In Proc. of 1993 ACM Symposium on Principles of Database Systems.
12. M.T. Goodrich, J.-J. Tsay, D.E. Vengroff, J.S. Vitter: External-Memory Computational Geometry. In Proc. of 34th IEEE Foundations of Computer Science, 1993.
13. S. Malik, A.R. Wang, R.K. Brayton, A. Sangiovanni-Vincentelli: Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In Proc. of 1988 IEEE International Conference on CAD.
14. H. Ochi, N. Ishiura, S. Yajima: Breadth-First Manipulation of SBDD of Boolean Functions for Vector Processing. In Proc. of 28'th ACM/IEEE Design Automation Conference, 1991.
15. H. Ochi, K. Yasuoka, S. Yajima: Breadth-First manipulation of Very Large Binary-Decision Diagrams. In Proc. of 1993 IEEE International Conference on CAD.
16. R. Rudell: Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In Proc. of 1993 IEEE International Conference on CAD.
17. C. Ruemmler, J. Wilkes: An introduction to disk drive modeling. IEEE Computer, 27 (3), 1994.
18. D. Sieling, I. Wegener: Reduction of OBDDs in linear time. Information Processing Letters, 48, 1993.
19. D.E. Vengroff, J.S. Vitter: I/O-Efficient Scientific Computation Using TPIE. In Proc. of 7th IEEE Symposium on Parallel and Distributed Processing, 1995.
20. J.S. Vitter, E.A.M Shrive: Algorithms for Parallel Memory I: Two-level Memories. Algoritmica, 12 (2), 1994.