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

Lars Arge

BRICS Report Series RS-96-29

ISSN 0909-0878 August 1996

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

Lars Arge†

BRICS‡
Department of Computer Science
University of Aarhus
Aarhus, Denmark

August 1996

**Abstract**

Ordered Binary-Decision Diagrams (OBDD) are the state-of-the-art data structure for boolean function manipulation and there exist several software packages for OBDD manipulation. OBDDs have been successfully used to solve problems in e.g. digital-systems design, verification and testing, in mathematical logic, concurrent system design and in artificial intelligence. The OBDDs used in many of these applications quickly get larger than the avaliable main memory and it becomes essential to consider the problem of minimizing the Input/Output (I/O) communication. In this paper we analyze why existing OBDD manipulation algorithms perform poorly in an I/O environment and develop new I/O-efficient algorithms.

*An extended abstract version of this paper was presented at the Sixth International Symposium on Algorithms and Computation (ISAAC'95).

†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@brics.dk`

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

## 1 Introduction

Many problems in digital-systems design, verification and testing, mathematical logic, concurrent system design and artificial intelligence can be expressed and solved in terms of boolean functions [8]. The efficiency of such solutions depends on the data structures used to represent the boolean functions, and on the algorithms used to manipulate these data structures. Ordered Binary-Decision Diagrams (OBDDs) [7, 8] are the state-of-the-art data structure for boolean function manipulation and they have been successfully used to solve problems from all of the above mentioned areas. There exist implementations of OBDD software packages for a number of sequential and parallel machines [5, 6, 22, 23]. Even though there exist very different sized OBDD representations of the same boolean function, OBDDs in real applications tend to be very large. In [5] 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 OBDDs orders of magnitude larger. In such cases the Input/Output (I/O) communication becomes the bottleneck in the computation.

Until recently most research, both theoretical and practical, has concentrated on finding small OBDD representations of boolean functions appearing in specific problems [6, 8, 17, 24], or on finding alternative succinct representations while maintaining the efficient manipulation algorithms [13]. The limit on the size of the problem instances one has been able to solve in practice has generally been determined by the ability to find representations that fit in internal memory of the machine used to solve the problem. The underlying argument for concentrating on the problem of limiting the size of the OBDDs then seems to have been that as soon as they get to large—larger than the available main memory—generating a large number of page faults, resulting in dramatically increasing running times, is inevitable. Very recently however, researchers have instead begun to consider I/O issues arising when the OBDDs get larger than the available internal memory, and experimental results show that very large runtime speedups can be achieved with algorithms that try to minimize the access to external memory as much as possible [5, 23]. 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 speed. This will however just increase the significance of the I/O bottleneck since the development of disk technology lacks 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 [25].

In this paper we analyze why existing OBDD manipulation algorithms perform poorly in an I/O environment and develop new I/O-efficient algorithms.

## 1.1 I/O Model and Previous Results

We will be working in the *parallel disk model* [1, 31] which models the I/O system of many existing workstations. The model has the following parameters:

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

where $M < N$ and $1 \leq DB \leq M/2$. Depending on the size of the data elements, typical values for workstations and file servers in production today are on the order of $M = 10^6$ or $10^7$ and $B = 10^3$. Values of $D$ range up to $10^2$ in current disk arrays.

An I/O operation (or I/O) in the model is the process of simultaneously reading or writing $D$ blocks of data, one block of $B$ contiguous elements to or from each of the $D$ disks. The I/O-complexity of an algorithm is simply the number of I/Os it performs. Internal computation is free, and we always assume that the $N$ elements initially are stored in the first $N/DB$ blocks of each disk. Thus reading all the input data requires $N/DB$ I/Os. We will use the term *scanning* to describe the fundamental primitive of reading (or writing) all items in a set stored contiguously in external memory by reading (or writing) the blocks of the set in a sequential manner.

Early work on external-memory algorithms concentrated on sorting and permutation related problems [1, 11, 20, 21, 30, 31]. 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 [4, 14] and graph problems [10]. Other related papers are [27] and [12] that address the problem of computing the transitive closure of a graph under some restrictions on the size of the graph, and propose a framework for studying memory management problems for maintaining connectivity information and paths on graphs, respectively. Also worth noticing in this context is [18] that addresses the problem of storing graphs in a paging environment, but not the problem of performing computation on them, and [3] where a number of external (batched) dynamic data structures are developed. Finally, it is demonstrated in [9, 29] that the results obtained in the mentioned papers are not only of theoretical but also of great practical interest.

While $N/DB$ is the number of I/Os needed to read all the input,[1] Aggarwal and Vitter [1] proved that $\Theta(\frac{N}{DB}\log_{M/B}\frac{N}{B}) = \Theta(\text{sort}(N))$[2] is the number of I/Os needed to sort $N$ elements. Furthermore, they proved that the number of I/Os needed to rearrange $N$ elements according to a given permutation is $\Theta(\min\{N/D, \text{sort}(N)\}) = \Theta(\text{perm}(N))$.[2] They also developed algorithms with I/O bounds matching these lower bounds in the $D = 1$ model. Later the results have been extended to the general model [20, 19, 30, 31]. In [10] it was shown that the permutation lower bound also applies to a large number of fundamental graph problems.

Taking a closer look at the fundamental bounds for typical values of $B$ and $M$ reveals that because of the large base of the logarithm, $\log_{M/B}\frac{N}{B}$ is less than 3 or 4 for all realistic values of $N, M$ and $B$. This means that the sorting bound in all realistic cases will be smaller than $N/D$, such that $\text{perm}(N) = \text{sort}(N)$. In practice the term in the bounds that really makes the difference is the $DB$-term in the denominator of the sorting bound. As typical values of $DB$ are measured in thousands, going from a $\Omega(N)$ bound—as we shall see is the worst-case I/O performance of many internal-memory algorithms—to the sorting bound, can be really significant in practice.

## 1.2 OBDDs and Previous Results

An OBDD is a *branching program* with some extra constraints. A branching program is a directed acyclic graph with one root, whose leaves (sinks) are labeled with boolean constants. The non leaves are labeled with boolean variables and have two outgoing edges labeled 0 and 1, respectively. If a vertex is labeled with the variable $x_i$ we say that it has index $i$. If $f$ is the

[1]We refer to $N/DB$ as the *linear* number of I/Os.

[2]For simplicity we write $\text{sort}(N)$ and $\text{perm}(N)$, suppressing $M, B$ and $D$.

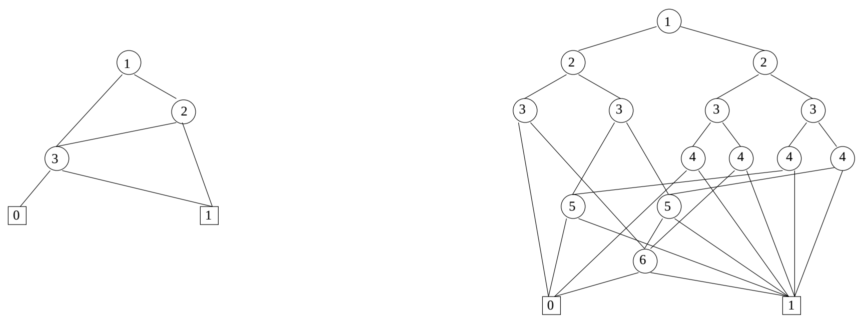


Figure 1: OBDD representations of the functions $x_1 \wedge x_2 \vee x_3$ and $x_1 \wedge x_4 \vee x_2 \wedge x_5 \vee x_3 \wedge x_6$. The left children are 0-successors and the right 1-successors. All edges are directed downwards.

Boolean function represented by the branching program, an evaluation of $f(a_1, \ldots, a_n)$ starts at the root and follows for a vertex labeled $x_i$ the outgoing edge with label $a_i$. The label of the sink reached in this way equals $f(a_1, \ldots, a_n)$. 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, $x_1, \ldots, x_n$, that is, 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. Figure 1 shows two examples of OBDDs. 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 minimizes the OBDD representation of a given function [17, 24].

In [7] 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. Bryant also proved that iterated use of the following two reduction rules on an OBDD with at most one 0-sink and one 1-sink 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. The OBDDs in Figure 1 are both reduced. Bryant [7] gave an algorithm for reducing an OBDD $G$ with $|G|$ vertices in $O(|G| \log |G|)$ time. Later algorithms running in $O(|G|)$ time have been developed [8, 26].

The most fundamental operations on OBDDs are the following:

- Given an OBDD representing $f$, compute if $f$ can be satisfied.
- Given two OBDDs representing $f_1$ and $f_2$, compute if $f_1 = f_2$.
- Compute from OBDDs representing $f_1$ and $f_2$ an OBDD for $f = f_1 \otimes f_2$, where $\otimes$ is some boolean operator.

The first two operations can easily be performed on *reduced* OBDDs. From a computational point of view the fundamental operations are therefore the reduce operation and the apply operation, as we shall call the operation which computes the reduced OBDD for the function obtained by combining two other functions by a binary operator. In [7] an $O(|G_1| \cdot |G_2|)$ time algorithm for using the apply operation on two OBDDs of size $|G_1|$ and $|G_2|$ is developed. This algorithm relies on a depth first traversal algorithm. In [22] a breadth first traversal algorithm with the same time bound is given.

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 [5, 6, 7, 8, 23], it was only very recently that OBDD manipulation algorithms especially designed to minimize I/O were developed. In [23] Ochi, Yasuoka and Yajima realized that the traditional depth first and breadth first apply algorithms do not perform well when the OBDDs are too large to fit in internal memory, and they developed alternative algorithms working in a levelwise manner.[3] These algorithms were obtained by adding extra vertices to the representation (changing the OBDD definition) such that the index of successive vertices on any path in an OBDD differs by exactly one. This makes the previously developed breadth first algorithm [22] work in a levelwise manner. Implementing the algorithms they obtained runtime speedups of several hundreds compared to an implementation using depth first algorithms. Very recently Ashar and Cheong [5] showed how to develop levelwise algorithms without introducing extra vertices, and conducted experiments which showed that on large OBDDs their algorithms outperform all other known algorithms. As the general idea (levelwise algorithms) are the same in [22] and [5], we will only consider the latter paper here. Finally, it should be mentioned that Klarlund and Rauhe in [16] report

[3]When we refer to depth first, breadth first and levelwise algorithms we refer to the way the apply algorithm traverse the OBDDs. By levelwise algorithm we mean an algorithm which processes vertices with the same index together. All known reduce algorithms work in a levelwise manner.

significant runtime improvements when working on OBDDs fitting in internal memory and using algorithms taking advantage of the blocked transport of data between cache and main memory.

In the algorithms in [5] no explicit I/O control is used. Instead the algorithms use a virtual memory space and the I/O operations are done implicitly by the operation system. However, explicit memory management is done in the sense that memory is allocated in chunks/blocks that match the size of an I/O block, and a specific index is associated with each such block. Only vertices with this index are then stored in such a block. This effectively means that the OBDDs are stored in what we will call a level blocked manner. The general idea in the manipulation algorithms is then to try to access these level blocked OBDDs in such a way that the vertices are accessed in a pattern that is as levelwise as possible. On the other hand we in this paper assume that we can explicitly manage the I/O. This could seem to be difficult in practice and time consuming in terms of internal computation. However, as Vengroff and Vitter show in [29]—using the transparent parallel I/O environment (TPIE) developed by Vengroff [28]—the overhead required to manage I/O can be made very small.

## 1.3 Our Results

In this paper we analyze why the "traditional" OBDD manipulation algorithms perform poorly when the OBDDs get large, by considering their I/O performance in the parallel disk model. Furthermore we develop new I/O-efficient algorithms.

First we show that all existing *reduce* algorithms—including the algorithms developed with I/O in mind [5]—in the worst case use $\Omega(|G|)$ I/Os to reduce an OBDD of size $|G|$. We show that this is even the case if we assume that the OBDD is blocked in external memory in some for the algorithm "natural" or favorable way by the start of the algorithm—depth first, breadth first or level blocked.[4] Then we show that for a special class of algorithms, which includes all existing algorithms, $\Omega(\text{perm}(|G|))$ is a lower bound on the number of I/Os needed to reduce an OBDD of size $|G|$. We show that this is even the case if we assume one of the blockings mentioned above, and even if we assume another intuitively good blocking. Previous I/O lower

[4]When we refer to a blocking as e.g. a depth first blocking, we refer to a blocking where the vertices are assigned to blocks in the way they are met in a depth first traversal.

bounds on graph problems all assume general blockings. Finally, we develop an $O(\text{sort}(|G|)$ I/O reduce algorithm. Thus our algorithm is asymptotically optimal in all realistic I/O-systems, among algorithms from the special class we consider.

We then go on and analyze the existing *apply* algorithms in the parallel disk model. Again we show that in the worst case all existing algorithms use $\Omega(|G_1| \cdot |G_2|)$ I/Os, and that this also holds for natural blockings of the involved OBDDs. We also develop an $O(\text{sort}(|R|))$ I/O apply algorithm. Here $|R|$ denotes the size of the resulting un-reduced OBDD. Our algorithm is thus asymptotically optimal in all realistic I/O-systems assuming that we have to do a reduction step after the use of the apply algorithm.

We believe that the developed algorithms could be of enormous practical value, as the constants in the asymptotic I/O bounds are all small. As mentioned in [5] large runtime improvements open up the possibility of creating OBDDs for verifying very large portions of chips, something considered impossible until now.

The rest of the paper is organized with a section for each of the OBDD manipulation algorithms. For simplicity we only consider the one disk model in these two sections. In Section 4 we then discuss extension of our results to the general $D$-disk model. We end the paper with a concluding section.

## 2 The Reduce Operation

Our discussion of the reduce operation is divided into three main parts. In Subsection 2.1 we present the existing reduce algorithms in order to be able to analyze their I/O-behavior in Subsection 2.2. For natural reason these two subsection will be rather discussing, and not very mathematically strict as Subsection 2.3 where we prove a lower bound on the number of I/Os needed to reduce a given OBDD. Finally, we in Subsection 2.4 present our new I/O-efficient reduce algorithm.

In our discussions of reduce algorithms—and 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 indices (levels) of its two children. The same assumptions are made in the existing algorithms. 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 an id and an index (and maybe a pointer) for each of the two children. The vertices also contain a few other fields used by the apply and reduce algorithms.

## 2.1 Reduce Algorithms

All reduce algorithms reported in the literature basically works in the same way. In order to analyze their I/O behavior, we in this section sketch the basic algorithm and the different variations of it.

The basic reduce algorithm closely follows an algorithm for testing whether two trees are isomorphic [2]. The algorithm processes the vertices levelwise from the sinks up to the root, and tries to use the two reduction rules discussed previously on each vertex. When the root is reached the reduced OBDD has been obtained, as the reduction rules cannot be used on any of the vertices in the OBDD. More precisely the algorithm assigns an integer label to each vertex in the OBDD such that a unique label is assigned to each unique sub-OBDD: First, two distinct labels are assigned to the sink vertices—one to the 1-sinks and one to the 0-sinks—and then the vertices are labeled level by level (index by index). Assuming that all vertices with index greater than $i$ have been processed, a vertex $v$ with index $i$ is assigned a label equal to that of some other vertex that has already been relabeled, if and only if one of two conditions is satisfied (one of the two reduction rules can be used). First, if the labels of $v$'s children are equal the vertex is redundant and it is assigned a label equal to that of the children (R1). Secondly, if there exists some already processed vertex $w$ with index $i$ whose left and right children have the same labels as the left and right children of $v$, respectively, then the sub-OBDDs rooted in $v$ and $w$ is isomorphic, and $v$ is assigned the same label as $w$ (R2). When all vertices have been relabeled the OBDD consisting of precisely one vertex for each unique label is the reduced OBDD corresponding to the original one.

The reduction algorithm comes in three variants (disregarding I/O issues for now), and the main difference between them is the way they decide if reduction rule R2 can be used on a given vertex. When processing vertices with index $i$ Bryant's original algorithm [7] sorts the vertices according to the labels of their children such that vertices which should have assigned the same label end up next to each other in the sorted sequence. The time complexity of the algorithm is dominated by the time used to sort the vertices, such that the algorithm runs in time $O(|G| \log |G|)$. Later, Bryant [8] gave an algorithm which instead of sorting the vertices maintain a (hash) table with an entry for each unique vertex seen so far. When processing a vertex a lookup is made in the table to see if an isomorphic vertex has already been labeled. If not the vertex is given a new unique label and inserted in the table. The implementations reported in [5, 6, 22, 23] all use this general idea. From a theoretical point of view the table uses a lot of space, namely $O(n \cdot |G|^2)$, but using "lazy initialization" [2] the running time of the algorithm can be kept at $O(|G|)$, as that is the number of entries in the table which is actually used. Finally, the algorithm by Sieling and Wegener [26] also sorts the vertices with a given index according to the labels of the children, but uses the bounded size of the label domain to do so with two phases of the well-known bucket sort algorithm. First, the vertices are partitioned according to the label of the 0-successor, and in a second bucket sort phase the non-empty buckets are partitioned according to the 1-successor. Vertices that end up in the same bucket is then assigned the same label. The algorithm runs in $O(|G|)$ time.

## 2.2 The I/O behavior of Reduce Algorithms

The basic reduce algorithm by Bryant [7] starts by doing a depth first traversal of the OBDD in order to collect the vertices in lists according to their indices (levels). If one does not assume anything about the way the vertices are blocked—which probably is most realistic, at least if one works in a virtual memory environment and uses pointers to implement the OBDDs—an adversary can force the algorithm to use $\Omega(|G|)$ I/Os just to do this traversal: If we call the vertex visited as number $i$ in a depth first traversal for $v_i$, the adversary simply groups vertex $v_1, v_{M+1}, v_{2M+1}, \ldots, v_{(B-1)M+1}$ together into the first block, $v_2, v_{M+2}, v_{2M+2}, \ldots, v_{(B-1)M+2}$ into the second block, and so on. This results in a page fault every time a new vertex is visited. Even if one assumes that the OBDD is blocked in a breadth first manner, or even in a level manner, it is fairly easy to realize that the depth first traversal algorithm causes $\Omega(|G|)$ page faults in the worst case.

So let us assume that the OBDD is blocked in a depth first manner, such that the traversal can be performed in $O(|G|/B)$ I/Os. At first it seems that the algorithm still uses $\Omega(|G|)$ I/Os, as it during the traversal outputs the vertices to $n$ different lists (one for each index), and as an adversary can force it never to output two consecutive vertices in the depth first order to the same list. However, *in practice* this would not be to bad, as we typically have that $n \ll |G|$ and that $n$ actually is smaller than $M/B$, which means that we can reserve a block in internal memory for each of the $n$ lists and only do an output when one of these blocks runs full. Then we only use $O(|G|/B)$ I/Os to produce the lists. In general an algorithm which scans through the OBDD and distribute the vertices to one of $n$ lists will perform

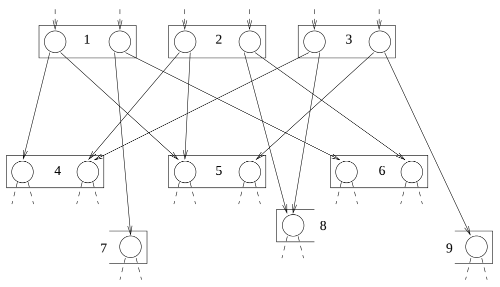


Figure 2: I/O behavior of reduce algorithms ($B = 2$).

well in practice. As we will discuss below this is precisely the idea used in the algorithm by Ashar and Cheong [5].

So let us then assume that we have produced the index lists (and thus a level blocking of the OBDD) in a acceptable number of I/Os, and analyze how the different variations of the basic algorithm then perform. Recall that all the variations basically sort the vertices with a given index according to the labels of their children. This means that when processing vertices with a given index the children have to be "visited" in order to obtain their labels. Assuming noting about the order in which this is done, it is not difficult to realize that an algorithm can be forced to do an I/O every time it visits a child. Actually, this holds whatever blocking one has—depth first, breadth first or level blocking—mainly because the vertices can have large fan-in. As an example of this, consider the part of an OBDD in Figure 2. We assume that the OBDD is level blocked and that $B = 2$ and $M/B = 3$, that is, that the main memory can hold 3 blocks. Now consider the process of visiting the children of each vertex in block 1 through 3, assuming that a least recently used (LRU) like paging strategy is used. First we load block 1 and start to visit the children of the leftmost vertex. To do so we load block 4 and then block 5. Then we continue to visit the children of the second vertex in block 1. To do so we have to make room for block 7, so we flush block 4 from internal memory. Then we can load block 7 and continue to load block 6, flushing block 5. This process continues, and it is easy to realize that we do an I/O every time we visit a vertex. Similar examples can be given for depth first and breadth first blockings.

The above problem is also realized in [5], and in order to avoid some of the randomness in the memory access the algorithm presented there visits the children in level order. This is accomplished by scanning through the vertices, distributing each of them to two of $n$ lists according to the index (level) of their children. As discussed above this can be done I/O-efficient in practice. Then these lists are processed one at a time and the desired labels are obtained. The algorithm follows the general philosophy mentioned earlier that as the vertices are stored levelwise they should also be accessed levelwise. But still it is not hard to realize that also this algorithm could be forced to do an I/O every time a child is accessed, because there is no correlation between the order in which the children on a given level are visited and the blocks they are stored in. For example using the strategy on the OBDD in Figure 2 still results in a page fault every time a child is visited. To summarize, all variations of the basic algorithm use $\Omega(|G|)$ I/O in the worst case to obtain the labels of the children—even the algorithm designed with I/O in mind. Furthermore, there seems to be no simple blocking strategy that avoids this.

Finally, there is the actual sorting step. It is difficult to say how many I/Os the basic algorithm uses on this task, as a number of different sorting algorithms could be used and as some of them might actually perform reasonably in an I/O environment. It is however easy to realize that the (hash) table approaches [5, 8] perform poorly on large OBDDs, as there is no regular pattern in the access to the table. Also the bucket approach used in [26] performs poorly because of the random pattern in which the (large number of) buckets are accessed.

To summarize, all known algorithms use $\Omega(|G|)$ I/Os in the worst case to reduce an OBDD of size $|G|$. As mentioned, this number could be very large compared to the linear or the sorting I/O bounds. There are several reasons why the algorithm in [5] still performs so relatively well in practice. We believe that the main reason is that the OBDDs used in the experiments in [5], even though they are large, still are small enough to allow one level of the OBDD to fit in internal memory. This, together with the intuitively correct levelwise blocking and access to the table, results in the large runtime speedups compared to other algorithms. A main reason is also to be found in the apply algorithm which we discuss in Section 3.

## 2.3 I/O Lower Bound on the Reduce Operation

After analyzing the I/O performance of existing reduce algorithms, we will now prove a lower bound on the number of I/Os we have to use in order to reduce an OBDD. As mentioned in the introduction, Aggarwal and Vitter [1] proved a lower bound on the number of I/Os needed to permute $N$

elements. They used a counting argument where they counted the number of permutations one can produce with a given number of I/Os and compared this to $N!$ We will use the same kind of technique to prove a lower bound on the reduce operation. However, while the current permutation is well defined throughout an algorithm for the permutation problem, this is generally not the case in graph computations like the reduce operation. In the permutation case one can regard the main and external memory as one big memory, and it is then easy to define the current permutation as the $N$ elements are all present somewhere in this memory throughout the algorithm. On the contrary elements (vertices) may disappear and new ones may be created during a reduce operation. In the extreme case all the vertices of an input OBDD are removed by a reduce operation and replaced by one (sink) vertex. In order to prove permutation-like bounds on graph problems, that is, bounds expressing the fact that in the I/O-model it is in general hard to rearrange elements according to a given permutation, we thus restrict our attention to a specific class of reduce algorithms. Intuitively, the class consists of all algorithms that work by assigning labels to vertices, and check if the reduction rules can be used on a vertex by checking the labels of its children. The assumption is that the children of a vertex are loaded into internal memory (if they are not there already) when their new label is checked. All known reduction algorithms belong to this class and the one we develop in Section 2.4 does as well. In order to define the class precisely, we in Section 2.3.1 define a pebble game played on a graph. We also discuss its relation to I/O-complexity. Then we in Section 2.3.2 consider a specific graph and prove a lower bound for playing the game on this graph. This result is then used to prove an I/O lower bound on the reduce operation. We prove the lower bound for a number of specific blockings.

### 2.3.1 The $(M, B)$-Blocked Red-Blue Pebble Game

In [15] Hung and Kung defined a red-blue pebble game played on directed acyclic graphs in order to define I/O-complexity. In their game there were no notion of blocks. Here we define a game which is also played on directed graphs with red and blue pebbles, but otherwise is rather different form the Hung and Kung game. Among other things our game takes blocks into account.

An mentioned our $(M, B)$-blocked red-blue pebble game is played on a graph. During the game the vertices of the graph hold a number of pebbles colored red or blue. The blue pebbles contain an integer each, called the *block index*. Also the edges of the graph will be colored blue or red. A *configuration* is a pebbled graph with colored edges. In the *start configuration* all vertices contain precisely one blue pebble and all edges are blue. Furthermore, precisely $B$ pebbles have the block index 1, precisely $B$ have block index 2, and so on up to $V/B$ (we assume without loss of generality that $B$ divides the number of vertices $V$). Throughout the game at most $M - B$ pebbles may be red. A *terminal configuration* is one where all pebbles are blue and all edges red. The rules of the game are the following:

Rule 1: (Input) Blue pebbles with the same block index may be colored red.

Rule 2: (Output) Up to $B$ red pebbles may be colored blue and given the same block index, while all other pebbles with that block index are removed from the game.

Rule 3: New red pebbles may be placed on any vertex with a red pebble.

Rule 4: The edge $(v_i, v_j)$ may be colored red if both vertices $v_i$ and $v_j$ contain at least one red pebble.

A *transition* in the game is an ordered pair of configurations, where the second one follows from the first one by using one of the above rules. A *calculation* is a sequence of transitions of which the first configuration is the start configuration. A calculation is *complete* if it ends with the terminal configuration. We define the *pebble I/O-complexity* of a complete calculation to be the number of transitions in the calculation defined by the use of rule one or two.

Playing the pebble game on a graph models an I/O algorithm with the graph as input, and pebble I/O-complexity corresponds to I/O-complexity as defined in the introduction. Blue pebbles reflect vertices stored on disk and red pebbles vertices stored in main memory. In the start configuration the graph is stored in the first $V/B$ blocks on disk. Rule one and two then correspond to an input and output respectively, while rule three allows copying of vertices in internal memory (and thus storing of the same vertex in different blocks on disk). Finally, rule four—together with the definition of terminating configuration—defines the class of algorithms we want to consider, namely algorithms where for every edge $(v_i, v_j)$ in the graph, the algorithm at some point in the computation holds both vertices $v_i$ and $v_j$ in main memory at the same time.

Note that in the pebble game the external memory is divided into what is normally called *tracks*, as we read and write blocks of elements to or from a block of external memory with a unique block index. However, lower bounds proved in the pebble model also hold in the model discussed in the introduction, as an I/O reading or writing a portion of two tracks can be simulated with a constant number of I/Os respecting track boundaries.

### 2.3.2 Pebble I/O Lower Bound on the Reduce Operation

In [10] the following generalization of the permutation lower bound from [1] is proved:

**Lemma 1** *Let A be an algorithm capable of performing* $(N!)^{\alpha}N^{c}$ *different permutations on an input of size* $N$*, where* $0 < \alpha \leq 1$ *and* $c$ *are constant. Then at least one of these permutations requires* $\Theta(\mathrm{perm}(N))$ *I/Os.*

Using this lemma an $\Omega(\mathrm{perm}(N))$ lower bound can be shown on the number of I/Os needed to solve the *proximate neighbors problem* [10]. The proximate neighbors problem is defined as follows: Initially, we have $N$ elements in external memory, each with a key that is a positive integer $k \leq N/2$. Exactly two elements have each possible value of $k$. The problem is to permute the elements such that, for every $k$, both elements with $k$ are in the same block. In [10] the proximate neighbors problem is used to prove lower bounds on a number of important graph problems. We define a variant of the proximate neighbors problem called the *split proximate neighbors problem* (SPN). 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. Furthermore, we require that the keys of the first $N/2$ elements are sorted. Following the proof of the lower bound on the proximate neighbors problem we can prove the following:

**Lemma 2** *Solving SPN requires* $\Omega(\mathrm{perm}(N))$ *I/Os in the worst case, that is, there exists an instance of SPN requiring* $\Omega(\mathrm{perm}(N))$ *I/Os.*

*Proof*: There are $(N/2)!$ distinct split proximate neighbors problems. We define a block permutation to be an assignment of elements to blocks. For each of the distinct problems an algorithm will do some permutation in order to reach a block permutation that solves it. We want to estimate how many distinct problems one block permutation can be solution to. Consider the first block of a given block permutation. This block contains $B/2$ elements from the first part of the split proximate neighbors problem and $B/2$ elements from the last part. The elements have precisely $B/2$ different keys $k_1, k_2, \ldots, k_{B/2}$. Now let $i_1, i_2, \ldots, i_{B/2}$ be the indices of the elements from the last half of the problem, that is, the positions of the elements in the input configuration. The block permutation in hand can only be a solution to problems in which the keys $k_1, k_2, \ldots, k_{B/2}$ are distributed among the elements with indices $i_1, i_2, \ldots, i_{B/2}$ in the start configuration. This can be done in $(B/2)!$ different ways. This holds for all the $N/B$ blocks in the block permutation, and therefore $((B/2)!)^{N/B}$ is an upper bound on the number of distinct problems one block permutation can be a solution to. Thus we have that $\frac{(N/2)!}{((B/2)!)^{N/B}}$ is a lower bound on the number of block permutations an algorithm solving SPN must be able to perform. As $\frac{(N/2)!}{((B/2)!)^{N/B}} = \Omega\left(\frac{(N!)^{1/3}}{((B/2)!)^{N/B}}\right)$, and as we can rearrange the elements within each block of a block permutation in an additional $N/B$ I/Os, the algorithm could produce $(N!)^{1/3}$ permutations. The bound then follows from Lemma 1. □

Using SPN we can now prove a lower bound on the number of pebble I/Os needed to complete a specific pebble game. Lemma 2 tells us that there exists at least one SPN instance $X$ of size $N$ which requires $\Omega(\text{perm}(N))$ I/Os. We can now obtain an algorithm for $X$ from a pebble game by imagining that the elements in $X$ are written on some of the pebbles in a specific graph. Figure 3 shows how we imagine this encoding. The marked vertices are the ones containing elements from $X$, and the vertices to the left of the vertical dotted line contain the first half of the elements. Vertices containing elements

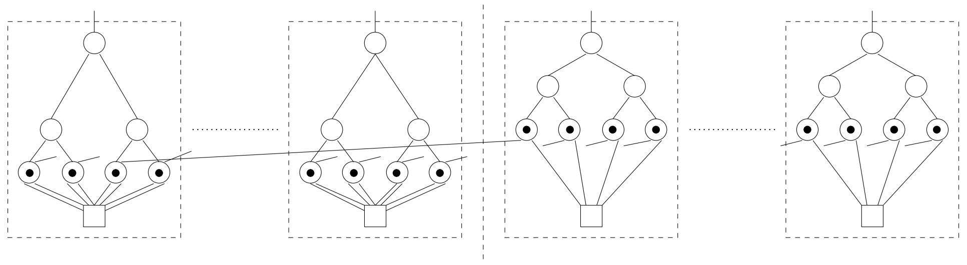


Figure 3: Graph used to obtain SPN algorithm from pebble game ($B = 8$).

with the same key are connected with an edge. In the start configuration the pebbles in the dotted boxes (blocks) have the same block identifier.[5] We can now prove the following:

**Lemma 3** *Completing the pebble game on the graph in Figure 3 takes at least* $\Omega(\mathrm{perm}(N))$ *pebble I/Os.*

*Proof*: Any sequence of transitions $S$ that completes the game can be used to construct an I/O algorithm for the hard SPN instance $X$ (or rather a sequence of I/Os that solve $X$). The I/O algorithm first scans through $X$ to produce a configuration where the first block contains the first $B/2$ input elements, the next the next $B/2$ elements and so on. This is done in $O(N/B)$ I/Os. Then it simulates the pebble I/Os in $S$, that is, every time a rule one transition is done in $S$ involving vertices used in the encoding of $X$, the algorithm performs a similar input, and every time a rule two transition involving vertices used in the encoding is done it performs a similar output. Now every time an edge between vertices used in the encoding is colored red in the game (using a rule four transition), two elements with the same key in $X$ are in main memory. When this happens the I/O algorithm puts the two elements in a special block in main memory. We have such a spare block because the pebble game was designed only to use $M - B$ internal memory. The block is written back to disk by the algorithm when it runs full. Thus when the pebble game is complete the I/O algorithm will have produced a set of blocks on disk solving the SPN problem on instance $X$. However there is one complication, as the pebble game allows copying of elements. We did not allow copying of elements in the SPN lower bound proof, so the solution to $X$ should consist of the original elements on not of copies. But given the sequence of I/Os solving $X$ produced above, we can easily produce another sequence solving $X$ which do not copy elements at all, simply by removing all elements except for those constituting the final solution. As the number of I/Os performed by the algorithm is bounded by $O(N/B)+O(|S|)$ the lemma follows. $\square$

Having proved a lower bound on the number of pebble I/Os needed to complete a $(M, B)$-blocked pebble game on the graph in Figure 3, we can now easily prove a lower bound on the number of pebble I/Os needed by a reduce operation. First we build a complete tree on top of the *base blocks*

[5]We assume without loss of generality that 2 and $B$ divide $N$.

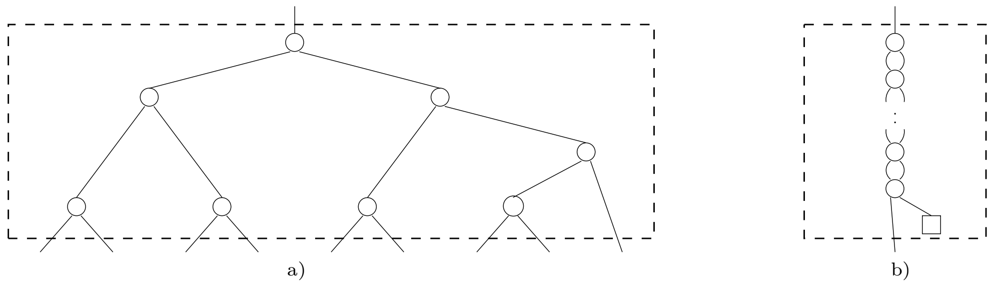


Figure 4: a) One block of top blocking in the breadth first blocking lower bound ($B = 8$). b) One of the "fill blocks" in proof of Lemma 6 and 7.

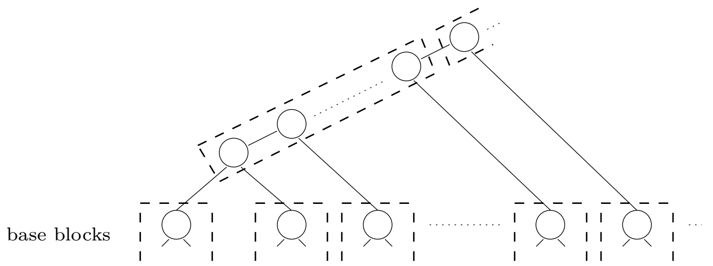


Figure 5: Top-blocking in the depth first blocking lower bound.

as we shall call the blocks in Figure 3. The tree is blocked as pictured in Figure 4a), and we obtain a breadth first blocked OBDD containing $O(N)$ vertices. In a similar way we can obtain a depth first blocked OBDD of size $O(N)$ by building the tree pictured in Figure 5 on top of the base blocks. As Lemma 3 also holds for these extensions of the graph in Figure 3, and as reducing such a graph completes the pebble game on the graph, we get the following.

**Lemma 4** *Reducing a breadth first or depth first blocked OBDD with* $|G|$ *vertices requires* $\Omega(\text{perm}\,|G|)$ *pebble I/Os in the worst case.*

Recall that the OBDDs in [5] are level blocked. It is easy to repeat the above proof for a level blocked OBDD and thus Lemma 4 also holds for such blockings. But proving these lower bounds does not mean that we could not be lucky and be able to reduce an OBDD in less I/Os, presuming that it is blocked in some other smart way.[6] However, when we later consider the

[6]We would then of course also have to worry about maintaining such a blocking between operations.

apply operation it turns out that the blockings we have considered (depth, breadth and level) are in fact the natural ones for the different algorithms for this operation. Intuitively however, the best blocking strategy for the class of reduce algorithms we are considering, would be a blocking that minimizes the number of pairs of vertices connected by an edge which are not in the same block—what we will call a *minimal-pair blocking*. But as we will prove next, a slightly modified version of the breadth first blocking we just considered is actually such a minimal-pair blocking. Thus the lower bound also holds for minimal-pair blockings. The modification consists of inserting a layer of the blocks in Figure 4b) between the base blocks and the blocked tree on top of them. The blocks are inserted purely for "proof-technical" reasons, and the effect of them is that every path of length less than $B$ between a marked vertex in the left half and a marked vertex in the right half of the base blocks must contain one of the edges between marked vertices.

In order to prove that the blocking of the graph $G$ in Figure 3 and 4 is indeed a minimal-pair blocking, we first state the following lemma which follows directly from the fact that every vertex in $G$ has at least in-degree one.

**Lemma 5** *For all blockings of $G$ every block must have at least one in edge.*

Now intuitively the blocking in Figure 3 and 4 is a minimal-pair blocking because all blocks, except for the base blocks, have one in edge, and because the vertices in the base blocks cannot be blocked in a better way than they are, that is, they must result in at least half as many edges between vertices in different blocks as there are marked vertices—we call such edges *pair breaking* edges. We will formalize and prove this in the next series of lemmas.

Call the first $N/2$ marked vertices in Figure 3 for $a$-vertices, and the last $N/2$ marked vertices for $b$-vertices. These vertices are the "special" vertices, because they have edges to each other and to the multi fan-in sink vertices (we call the sink vertices $c$-vertices). We now consider a block in an arbitrary blocking of $G$ containing one or more of these special vertices.

**Lemma 6** *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, such that $a_1, a_2, b_1$ and $b_2$ are all $\leq B/2$, and such 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.*

*Proof*: First we assume without loss of generality that all $a_1$ $a$-vertices are from the same base block. This is the best case as far as pair breaking edges are concerned, because they are all connected to the same $c$-vertex. Similarly, we assume that all $b_1$ $b$-vertices are from the same base block. Finally, we make the same assumption for the $a_2$ $a$-vertices and the $b_2$ $b$-vertices without their corresponding $c$-vertices. Making these assumptions we are left with vertices from at most four base blocks. If we can prove the lemma for this particular configuration it also holds for all other configurations.

We divide the proof in cases according to the value of $a_1$ and $b_1$:

- $a_1 = 0$

  In this configuration every $a$-vertex accounts for at least two pair breaking edges—namely the two edges to the $c$-vertices corresponding to the $a_2$ $a$-vertices. But then it is easy to realize that $a$ and $b$ vertices accounts for $a_2 + b_1 + b_2$ pair breaking edges. One of the edges of the $a$ vertices accounts for $a_2$ of them. For every $b$-vertex its corresponding $a$-vertex is either not one of the $a_2$ $a$-vertices, in which case the $b$-vertex itself accounts for a pair-breaking edge, or it is one of the $a$-vertices, in which case the other of the $a$-vertices pair breaking edges with a $c$ vertex can be counted. Finally, because of the "fill blocks" above the base blocks (Figure 4), we cannot connect the base blocks except with edges between $a$ and $b$ vertices. This means that every base block must contribute with one pair breaking edge not counting $a$-$b$ edges. This gives the extra $k$ pair breaking edges.

- $a_1 \geq 1$,$b_1 \geq 1$

  To hold $a$ and $b$-vertices and the two $c$-vertices we use $a_1+a_2+b_1+b_2+2$ of $K$'s capacity of $B$ vertices. The number of pair breaking edges in a block consisting of *only* these vertices is $a_1 + 2(B/2 - a_1) + 3a_2 + b_1 + (B/2 - b_1) + 2b_2 + |(a_1 + a_2) - (b_1 + b_2)|$. The first two terms correspond to the $a_1$ $a$-vertices and the third term to the $a_2$ $a$-vertices—not counting edges between $a$ and $b$ vertices. Similarly, the next three terms correspond to the $b$-vertices. The last term counts the minimal number of pair breaking edges corresponding to edges between $a$ and $b$ vertices in the block (assuming that as many as possible are "paired").

  Now we add vertices one by one in order to obtain the final block $K$. We add them in such a way that when adding a new vertex, it has an edge to at least one vertex already in the block (if such a vertex exists). Because we cannot connect the base blocks except with $a$-$b$ edges (again because of the "fill blocks"), one added vertex can at most decrease the number of pair breaking edges by one. Assuming that every added vertex decreases the number of pair breaking edges by one, we end up with a block with $a_1 + 2(B/2 - a_1) + 3a_2 + b_1 + (B/2 - b_1) + 2b_2 + |(a_1 + a_2) - (b_1 + b_2)| - (B - (a_1 + a_2 + b_1 + b_2 + 2)) = 3a_2 + 2b_2 + B/2 - a_1 + |(a_1 + a_2) - (b_1 + b_2)| + a_1 + a_2 + b_1 + b_2 + 2$ pair breaking edges. We want to prove that this number is at least $a_1 + a_2 + b_1 + b_2 + k$, which is indeed the case if $3a_2 + 2b_2 + B/2 - a_1 + |(a_1 + a_2) - (b_1 + b_2)| \geq k - 2$. This again trivially holds because $a_1 \leq B/2$ and because $k$ is only greater than 2 if $a_2$ and/or $b_2$ is non zero.

- $a \geq 1, b_1 = 0$

  As $b_1 = 0$, $b_2$ must be non zero. Doing the same calculations as in the previous case we find that the lemma holds if $3a_2 + 2b_2 - a_1 + |a_1 + a_2 - b_2| \geq k - 1$.

  Now if $a_1 + a_2 \geq b_2$ we get that $4a_2 + b_2$ should be grater than or equal to $k - 1$, which is trivially fulfilled as $k = 2$ if $a_2 = 0$ and 3 otherwise.

  If $a_1 + a_2 \leq b_2$ we get that $2a_2 + 3b_2 - 2a_1$ should be greater than or equal to $k - 1$. This is again trivially fulfilled (under the assumption $a_1 + a_2 \leq b_2$).

□

Now we want to remove the $a_1, a_2, b_1, b_2 \leq B/2$ assumption from Lemma 6. In the proof the assumption was used to easily be able to bound the number of pair breaking edges between $c$-vertices in $K$ and $a$ and $b$-vertices outside the block. Note that if one of the variables is greater than $B/2$ then the others must be less than $B/2$.

**Lemma 7** *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, such 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.*

*Proof*: As in the proof of Lemma 6 we assume that all (or as many as possible) $a$ and $b$-vertices of the same type are in the same base block. Taking a closer look at the proof of Lemma 6 quickly reveals that the proof works even if $a_2$ or $b_2$ is greater than $B/2$. The proof also works if $a_1 = 0$, so we are left with the following two cases:

- $b_1 > B/2$ (and $1 \leq a < B/2$)

  The block consisting of only $a$ and $b$-vertices and the corresponding $c$-vertices have at least $a_1 + 2(B/2 - a_1) + 3a_2 + b_1 + (B/2 - (b_1 - B/2)) + 2b_2 + |(a_1 + a_2) - (b_1 + b_2)| = 3a_2 + 2b_2 + 2B - a_1 + |(a_1 + a_2) - (b_1 + b_2)|$ pair-breaking edges. Assuming that the rest $(B - (a_1 + a_2 + b_1 + b_2 + 3))$ of the vertices in the block all brings the number of pair breaking edges down by one, the lemma follows from the fact that the following inequality is trivially satisfied $3a_2 + 2b_2 + B - a_1 + |(a_1 + a_2) - (b_1 + b_2)| \geq k - 3$.

- $a_1 > B/2$

  We divide the proof in two:

  - $b_1 \geq 1$

    The number of pair breaking edges "produced" by $a$ and $b$-vertices is at least $a_1 + 2(B/2 - (a_1 - B/2)) + 3a_2 + b_1 + (B/2 - b_1) + 2b_2 + |(a_1 + a_2) - (b_1 + b_2)|$ leading to the following inequality $3a_2 + 2b_2 + 3/2B - a_1 + |(a_1 + a_2) - (b_1 + b_2)| \geq k - 3$. This inequality is trivially satisfied as $a_1 < B$.

  - $b_1 = 0$ (and $b_2 \geq 1$)

    Using the same argument the satisfied inequality is $3a_2 + 2b_2 + B - a_1 + |(a_1 + a_2) - (b_1 + b_2)| \geq k - 3$.

□

We can now prove the main lemma:

**Lemma 8** *The breadth first blocking of G in Figure 3 and 4 is a minimal-pair blocking.*

*Proof*: First note that Lemma 7 also holds (trivially) if a block only contains vertices of one type. It then follows that every $a$ and $b$-vertex must result in at least one pair breaking edges. Furthermore, every $c$-vertex (or rather every base block) must result in at least one additional pair breaking edge.

Thus the blocking in Figure 3 obtains the minimal number of pair breaking edges. From Lemma 5 we furthermore know that every other block must have at least one in edge. The lemma then follows from the fact that the blocking in Figure 4 only has one in edge. □

Now we have proved that the intuitively best blocking for the reduction algorithm, as well as all the intuitively best blockings for the apply algorithms, all result in a lower bound of $\Omega(\mathrm{perm}(N))$ I/Os on the reduce operation. The results can be summarized as follows.

**Theorem 1** *Reducing an OBDD with $|G|$ vertices— depth first, breadth first, minimal-pair or level blocked—requires $\Omega(\mathrm{perm}(|G|))$ pebble I/Os in the worst case.*

## 2.4 I/O-Efficient Reduce Algorithm

Recall that one of the main problems with existing reduce algorithms with respect to I/O is that when they process a level of the OBDD they perform a lot of I/Os in order to get the labels of the children of vertices on the level. Our solution to this problem is simple—when a vertex is given a label we "inform" all its immediate predecessors about it in a "lazy" way using an external priority queue developed in [3]. On this priority queue we can do a sequence of $N$ insert and deletemin operations in $O(\mathrm{sort}(N))$ I/Os in total. After labeling a vertex we thus insert an element in the queue for all predecessors of the vertex, and we order the queue such that we on higher levels simply can perform deletemin operations to obtain the required labels.

In order to describe our algorithm precisely we need some notation. We refer to a vertex and its label with the same symbol (e.g. $v$). The index or level of a vertex is referred to as $id(v)$, and the 0-successor and 1-successor are referred to as $low(v)$ and $high(v)$, respectively. In order to make our reduce algorithm I/O-efficient, we start the algorithm by creating two sorted lists of the vertices in the OBDD we want to reduce. The first list (L1) contains the vertices sorted according to index and secondary according to label—that is, according to $(id(v), v)$. Put another way, the list represents a level blocking of the OBDD. The second list (L2) contains two copies of each vertex and it is sorted according to the index of their children and secondarily according to the labels of their children. That is, we have two copies of vertex $v$ ordered according to $(id(low(v)), low(v))$ and $(id(high(v)), high(v))$, respectively. To create L1—for *any* blocking of the OBDD—we just scan through the vertices in the OBDD, inserting them in a priority queue ordered according to index and label, and then we repeatedly perform deletemin operations to obtain L1. Thus we use $O(\text{sort}(N))$ I/Os to produce L1. L2 can be produced in the same number of I/Os in a similar way.

We are now ready to describe our new reduce algorithm. Basically it works like all the other algorithms. We process the vertices from the sinks up to the root and assign a unique label to each unique sub-OBDD root. We start by assigning one label to all 0-sinks and another to all 1-sinks. This is done just by scanning through L1 until all sinks have been processed. During this process—and the rest of the algorithm—we output one copy of each unique vertex to a result list. After labeling the sink vertices we insert an element in the priority queue for each vertex that has a sink as one of its children. The elements contain the label of the relevant child (sink), and the queue is ordered according to level and label of the "receiving" vertex (the vertex having a sink as child). This is accomplished by merging the list of sinks with the appropriate (first) elements in L2. Now assume that we have processed all vertices above level $i$ and want to process level $i$. In the priority queue we now have one or two elements for each vertex that has a child on a lower level than $i$. In particular we have two elements for every vertex on level $i$. Because the elements in the priority queue are ordered according to $(id(v), v)$, we can thus just do deletemin operations on the queue until we have obtained the elements corresponding to vertices on level $i$. At the same time we merge the elements with L1 in order to "transfer" the labels of the children to the relevant vertices in L1. Then we proceed like Bryant [7]; we sort the vertices according to the labels of the children (with an I/O optimal sorting algorithm [1, 31]), and use the reduction rules to assign new labels. Then we sort the vertices back into their original order, merge the resulting list with L2, and insert the appropriate elements (vertices with a child on level $i$) in the priority queue—just like after assigning labels to the sinks. When we reach the root we have obtained the reduced OBDD.

In order to analyze the I/O use of the algorithm, note that a linear number of operations in the size of the OBDD is performed on the priority queue. Thus we in total use $O(\text{sort}(|G|))$ I/Os to manipulate the queue. The I/O use of the rest of the algorithm is dominated by the sorting of the elements on each level, that is, by $O(\text{sort}(|G|))$ in total. We then have:

**Theorem 2** *A $|G|$ vertex OBDD $G$ can be reduced in $O(\text{sort}(|G|))$ I/Os.*

# 3 The Apply Operation

We divide our discussion of the apply operation into subsections on existing algorithms, their I/O-behavior, and on our new I/O-efficient algorithm.

## 3.1 Apply Algorithms

The basic idea in all the apply algorithms reported in the literature is to use the 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 $b$. Using this formula Bryant's algorithm [7] works as follows: Consider two functions $f_1$ and $f_2$ represented by OBDDs with roots $v_1$ and $v_2$. First, suppose both $v_1$ and $v_2$ are sinks. Then the resulting OBDD consists of a sink having the boolean value $value(v_1) \otimes value(v_2)$. Otherwise, suppose that at least one of the vertices is not a sink vertex. If $id(v_1) = id(v_2) = i$ the resulting OBDD consists of a root vertex with index $i$, and with the root vertex in the OBDD obtained by applying the apply operation on $low(v_1)$ and $low(v_2)$ as 0-child and with the root vertex in the OBDD obtained by applying the apply operation on $high(v_1)$ and $high(v_2)$ as 1-child. Thus a vertex $u$ with index $i$ is created and the algorithm is used recursively twice to obtain $low(u)$ and $high(u)$. Suppose on the other hand (and without loss of generality) that $id(v_1) = i$ and $id(v_2) > i$. Then the function represented by the OBDD with root $v_2$ is independent of $x_i$ (because of the fixed variable ordering), that is, $f_2|_{x_i=0} = f_2|_{x_i=1} = f_2$. Hence, a vertex $u$ with index $i$ is created, and the algorithm is recursively applied on $low(v_1)$ and $v_2$ to generate the OBDD whose root becomes $low(u)$, and on $high(v_1)$ and $v_2$ to generate the OBDD whose root becomes $high(u)$. This is basically the algorithm except that 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—dynamic programming is used: During the algorithm a table of size $|G_1| \cdot |G_2|$ is maintained. The $xy$'th entry in this table contains the result (the label of the root vertex) of using the algorithm on the vertex in the OBDD for $f_1$ with label $x$ and the vertex in the OBDD for $f_2$ with label $y$, *if* it is already computed. Before applying the algorithm to a pair of vertices it is checked whether the table already contains an entry for the vertices in question. If that is the case the result already computed is just returned. Otherwise, the algorithm continues as described above and adds the root vertex to the table. It is straightforward to realize that Bryant's algorithm runs in $O(|G_1| \cdot |G_2|)$ time (the size of the dynamic programming table). Note that it is proved in [7] that there actually exist reduced OBDDs representing functions $f_1$ and $f_2$ such that the size of $f_1 \otimes f_2$ is $\Theta(|G_1| \cdot |G_2|)$.

Due to the recursive structure Bryant's algorithm works in a depth first manner on the involved OBDDs. In [22] an algorithm algorithm working in a breadth first manner is developed in order to perform OBDD manipulation efficiently on a CRAY-type supercomputer. This algorithm works like Bryant's, except that recursive calls (vertices which need to have their children computed—we call them *requests*) are inserted in a queue (the *request* queue) and computed one at a time. This leads to a breadth first traversal of the involved OBDDs. Also this algorithm uses dynamic programming and runs in $O(|G_1| \cdot |G_2|)$ time.

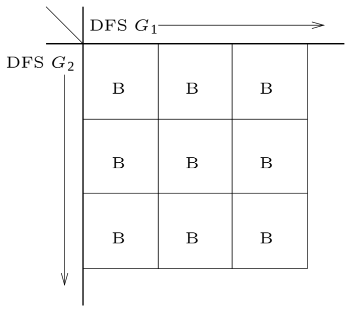
As previously discussed, I/O issues are then taken into account in [5] and an $O(|G_1| \cdot |G_2|)$ time algorithm working in a levelwise manner is developed. As in the case of the reduce algorithm it is assumed that the OBDDs are stored levelwise and the general idea is then to work as levelwise as possible on them. Basically the algorithm works like the breadth first algorithm, but with the request queue split up into $n$ queues—one for each level of the OBDDs. When a new request is generated it is placed in the queue assigned to the level of the vertex corresponding to the request. The queues are then processed one at a time from the queue corresponding to the top level and down. This way the OBDDs are traversed in a levelwise manner. Also *before* a new request is inserted in a queue it is checked if a duplicate request has already been inserted in the queue. This effectively means that the dynamic programming table and the request queues are "merged" into one structure. Finally, much like the way the reduce algorithm in [23] obtains the new labels of the children of a vertex in level order, the requests on a given level are handled in sorted order according to the levels of the requests issued as a consequence of them. As previously the motivation for this is that it assures that lookups for duplicate requests are done in a levelwise manner.

In order to obtain a canonical OBDD all the presented algorithms run the reduce algorithm after constructing a new OBDD with the apply operation. It should be noted that Bryant in [8] has modified his depth first algorithm such that the reduction is performed as an integrated part of the apply algorithm. The algorithm simply tries to use the reduction rules after returning from the two recursive calls. While it is easy to check if R1 can be applied, a table of the already generated vertices is used to check if R2 can be applied. The advantage of this modified algorithm is that redundant vertices, which would be removed in the following reduction step, is not created and thus space is saved. The algorithms not working in a depth first manner [5, 22, 23] cannot perform the reduction as an integrated part of the apply algorithm.

## 3.2 The I/O-Behavior of Apply Algorithms

Like we in Section 2.2 analyzed the existing reduce algorithms in the parallel disk model, we will now analyze the different apply algorithms in the model. In the following $|R|$ will be the size of the un-reduced OBDD resulting from a use of the apply algorithm on two OBDDs of size $|G_1|$ and $|G_2|$, respectively. We first analyze why the depth first [7] and breadth first [22] algorithms perform so poorly in an I/O-environment, and then we take a closer look at the algorithms developed with I/O in mind.

As in the case of the reduce algorithm it is not difficult to realize that assuming nothing about the blocking (which is probably the most realistic in practice) it is easy for an adversary to force both the depth first and the breadth first algorithm to do an I/O every time a new vertex in $G_1$ or $G_2$ is visited. This results in an overall use of $\Omega(|R|)$ I/Os, that is, $O(|G_1| \cdot |G_2|)$ I/Os in the worst case. It is equally easy to realize that breadth first and level blockings are just as bad for the depth first algorithm [7], and depth first and level blockings just as bad for the breadth first algorithm [22]. So let us assume that the OBDDs are blocked in some "good" way with respect to the used traversal scheme. Still the algorithms perform poorly because of the lack of locality of reference in the lookups in the dynamic programming table. To illustrate this we take a closer look at the depth first algorithm [7] assuming that the OBDDs are depth first blocked. Again, if we do not assume anything about the blocking of the dynamic programming table, it is easy to realize that in the worst case every access to the table results in a page fault. If we were able to block the table as we like, the only obvious way to block it would be in a depth first manner (Figure 6a): Assume that the algorithm is working on vertex $v_1$ in $G_1$ and $v_2$ in $G_2$. The algorithm now makes one of the following recursive calls: $v_1$ and $low(v_2)$, $low(v_1)$ and $v_2$, or $low(v_1)$ and $low(v_2)$. Before doing so it makes a lookup in the dynamic programming table. Thus the table should be blocked as indicated in Figure 6a) as we would like the corresponding part of the table to be in internal memory.



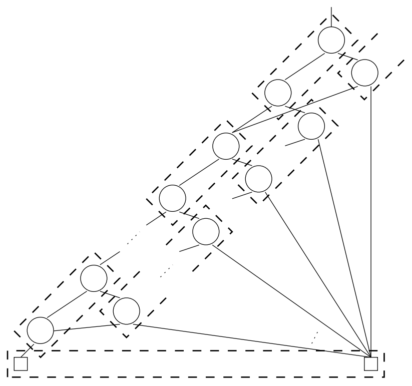


Figure 6: a) Dynamic programming table. b) I/O performance of algorithm in [5].

Note that with the blocking in Figure 6a) the algorithm would at least make a page fault on every $\sqrt{B}$ lookup operation. But actually it is much worse than that, which can be illustrated with the example in Figure 6b). Here a depth first blocking of an OBDD is indicated. It is also indicated how the 0-children of the "right" vertices can be chosen in an almost arbitrary way. This in particular means that an adversary can force the algorithm to make a lookup page fault every time one of these vertices is visited. As the number of such vertices is $\Theta(|G|)$ the algorithm could end up making a page fault for almost every of the $|R|$ vertices in the new OBDD.

After illustrating why the breadth first and depth first algorithms perform poorly, let us shortly consider the algorithm especially designed with I/O in mind [5]. Recall that this algorithm maintains a request queue for each level of the OBDDs, which also functions as the dynamic programming table divided into levels, and that it processes one level of requests in the order of the levels of the requests issued as a consequence of them. It is relatively easy to realize that in the worst case a page fault is generated every time one of the request queues is accessed as dynamic programming table. The reason is precisely the same as in the case of the depth first algorithm, namely that there is no nice pattern in the access to the table—not even in the access to one level of it. As previously, we therefore cannot block the queues efficiently and we again get the $\Omega(|R|)$ worst-case I/O behavior. The natural question to ask is of course why experiments then show that this approach can lead to the mentioned runtime speedups. The answer is partly that the traditional depth first and breadth first algorithms behave so poorly with respect to I/Os that just considering I/O issues, and actually try to block the OBDDs and access them in a "sequential" way, leads to large runtime improvements. Another important reason is the previously mentioned fact that in practical examples $n$ is much smaller than $M/B$, which means that a block from each of the $n$ queues fits in internal memory. However, we believe that one major reason for the experimental success in [5] 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, which again explains the good performance because the worst case behavior precisely occurs when one level does not fit in internal memory.

## 3.3 I/O-Efficient Apply Algorithm

The main idea in our new apply algorithm is to do the computation levelwise as in [5], but use a priority queue to control the recursion. Using a priority queue we do not need a queue for each level as in [5]. Furthermore, we do not check for duplicates when new requests are issued, but when they are about to be computed. Recall that the main problem with the previous levelwise algorithm precisely is the random lookups in the queues/tables when these checks are made. Instead of checking for duplicates when new requests are issued, we just insert them in a priority queue and perform the checks when removing requests from the queue. We do so simply by ordering the queue such that identical requests will be returned by consecutive deletemin operations. This way we can in a simple way ignore requests that have already been computed.

In order to make our algorithm work efficiently we need the vertices of each of the OBDDs sorted according to level and secondary according to label. This representation can easily be constructed in $O(\text{sort}(|G_1|) + \text{sort}(|G_2|))$ I/Os in the same way as we constructed the list L1 in the reduce algorithm. For convenience we now use four priority queues to control the recursion (instead of one). They all contain requests represented by a pair of vertices, one from $G_1$ and one from $G_2$. The first queue $V$ contains pairs $(v, w)$ where $id(v) < id(w)$, and is ordered according to the level and label of $v$. Thus $V$ contains requests which should be processed at level $id(v)$ and which can be processed without obtaining new information about

$w$. Similarly, the second queue $W$ contains pairs where $id(v) > id(w)$, ordered according to $(id(w), w)$. The last two priority queues $E_V$ and $E_W$ contain pairs where $id(v) = id(w)$, and are ordered according to $(id(v), v)$ and $(id(w), w)$, respectively.

We do the apply in a levelwise manner starting from the root vertices. We label the vertices in the resulting OBDD $R$ with pairs of labels from $G_1$ and $G_2$. The algorithm starts by comparing the indices of the two root vertices $v$ and $w$, and creates the root vertex $(v, w)$ of $R$ with index equal to the lowest of the two indices. If $id(v) < id(w)$ it then makes two new vertices $(low(v), w)$ and $(high(v), w)$ with indices $\min(id(low(v)), id(w))$ and $\min(id(high(v)), id(w))$, respectively, and "connects" $(v, w)$ to these two vertices. Similarly, if $id(v) > id(w)$ it makes the vertices $(v, low(w))$ and $(v, high(w))$, and if $id(v) = id(w)$ the vertices $(low(v), low(w))$ and $(high(v), high(w))$. Now the algorithm makes two recursive calls in order to construct the OBDDs rooted in the two newly created vertices. As the recursion is controlled by the priority queues this is done by inserting the vertices/requests in these queues. The level on which a given vertex/request $(u_1, u_2)$ is to be processed is determined by $\min(id(u_1), id(u_2))$. Therefore $(u_1, u_2)$ is inserted in $V$ if $id(u_1) < id(u_2)$ and in $W$ if $id(u_1) > id(u_2)$. If $id(u_1) = id(u_2)$ it is inserted both in $E_V$ and in $E_W$.

Now assume that the algorithm has processed all levels up to level $i - 1$. In order to process level $i$ we do the following: We do deletemin operations on $V$ in order to obtain all requests in this queue that need to be processed on level $i$. As discussed above we only process one copy of duplicate requests. As all requests $(u_1, u_2)$ in $V$ have $id(u_1) < id(u_2)$ all new requests generated as a consequence of them are of the form $(low(u_1), u_2)$ or $(high(u_1), u_2)$. Thus we do not need any new information about $u_2$ to issue the new requests. During the process of deleting the relevant requests from the queue we therefore simply "merge" the requests with the representation of $G_1$ in order to obtain the information needed. We process level $i$ requests in $W$ in a similar way. Finally, we process level $i$ requests in $E_V$ and $E_W$. We know that all vertices in these queues have $id(u_1) = id(u_2) = i$, which means that they will create requests of the form $(low(u_1), low(u_2))$ and $(high(u_1), high(u_2))$. Therefore we need to obtain new information from both $G_1$ and $G_2$. Thus we do deletemin operations on $E_V$ and "merge" the result with the representation of $G_1$, collecting the information we need from this OBDD. During this process we also insert the resulting vertices in $E_W$. Finally, we do deletemin operations on $E_W$ and "merge" with $G_2$ to obtain the information we need to issue the relevant new requests.

When the above process terminates we will have produced $R$, and the analysis of the I/O-complexity of the algorithm is easy: Each vertex/request is inserted in and deleted from a priority queue a constant number of times. Thus we directly obtain the following from the I/O bounds of the priority queue operations.

**Theorem 3** *The apply operation can be performed in* $O(\text{sort}(|R|))$ *I/Os.*

## 4 Extension of Results to $D$-disk Model

As promised we should make a few comments about our results in the $D$-disk model. As far as the lower bound is concerned, we can of course just divide our bound in the one-disk model by $D$ and the obtained bound will then be a lower bound in the parallel disk model. It turns out that this bound can actually be matched, that is, we can obtain a speedup proportional to the number of disks.

To obtain the upper bounds in this paper we only used three basic "primitives"; scanning, sorting and priority queues. Scanning through $N$ elements can easily be done in $O(N/DB)$ I/Os in the parallel disk model and as already mentioned we can also sort optimally in the model. Furthermore, it is proved in [3] that the priority queue can also take full advantage of parallel disks. Both the sorting algorithms and the priority queue on parallel disks work under the (non-restrictive in practice) assumption that $4DB \leq M - M^{1/2+\beta}$ for some $0 < \beta < 1/2$. Thus with the same assumption all the results obtained in this paper holds in the parallel disk model.

## 5 Conclusion and Open Problems

In this paper we have demonstrated how all the existing OBDD manipulation algorithms in the worst case make on the order of the number of memory accesses page faults. This is the reason why they perform poorly in an I/O environment. We have also developed new OBDD manipulation algorithms and proved their optimality under some natural assumptions.

We believe that the developed algorithms are not only of theoretical but also of practical interest—especially if we make a couple of modifications. If we represent the OBDDs in terms of edges instead of vertices (where each edge "knows" the level of both source and sink) and block them in the way they are used by the apply algorithm, it can be realized that our apply algorithm automatically produce the blocking used by the (following) reduce algorithm. The reduce algorithm can then again produce the blocking used by the (next) apply algorithm. This can be done without extra I/O use, basically because the apply algorithm works in a top-down manner while the reduce algorithm works in a bottom-up manner. With this modification the algorithms are greatly simplified and we save the I/Os used to create the special representations of the OBDDs used in the reduce and apply algorithms. Furthermore, it is also easy to realize that we can do with only one priority queues in the apply algorithm. As the constants in the I/O bounds on the priority queue operations are all small, the constants in the bounds of the developed OBDD manipulation algorithms are also small. Also it is demonstrated in [29] that the overhead required to explicitly manage I/O can be made very small, and therefore we believe that our algorithms could lead to large runtime speedups on existing workstations. We hope in the future to be able to implement the priority queue data structure in the Transparent Parallel I/O Environment (TPIE) developed by Vengroff [28] in order to verify this.

A couple of questions remains open, namely if it is possible to prove an $O(\text{perm}(N))$ I/O lower bound on the reduce operation assuming *any* blocking, and if it is possible to prove a lower bound on the apply operation without assuming that a reduce step is done after the apply.

## Acknowledgments

I would especially like to thank Peter Bro Miltersen for comments on drafts of this paper and for comments that lead to the definition of the blocked red-blue pebble game. I would also like to thank Allan Cheng for introducing me to OBDDs and Darren Erik Vengroff for inspiring discussions.

## References

[1] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[3] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995.

[4] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. Annual European Symposium on Algorithms, LNCS 979*, pages 295–310, 1995. A full version is to appear in special issue of Algorithmica.

[5] P. Ashar and M. Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *Proc. IEEE International Conference on CAD*, 1994.

[6] S. K. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. ACM/IEEE Design Automation Conference*, 1990.

[7] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[8] R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3), 1992.

[9] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.

[10] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[11] T. H. Cormen. *Virtual Memory for Data Parallel Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[12] E. Feuerstein and A. Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. In *Proc. Int. Symp. on Algorithms and Computation*, 1993.

[13] J. Gergov and C. Meinel. Frontiers of feasible and probabilistic feasible boolean manipulation with branching programs. In *Symposium on Theoretical Aspects of Computer Science, LNCS 665*, 1993.

[14] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.

[15] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. ACM Symp. on Theory of Computation*, pages 326–333, 1981.

[16] N. Klarlund and T. Rauhe. BDD algorithms and cache misses. Technical Report RS-96-26, BRICS, University of Aarhus, 1996.

[17] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. IEEE International Conference on CAD*, 1988.

[18] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[19] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 29–39, 1991.

[20] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 120–129, 1993.

[21] M. H. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. of the 26th Hawaii Int. Conf. on Systems Sciences*, 1993.

[22] H. Ochi, N. Ishiura, and S. Yajima. Breadth-first manipulation of sbdd of boolean functions for vector processing. In *Proc. ACM/IEEE Design Automation Conference*, 1991.

[23] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Proc. IEEE International Conference on CAD*, 1993.

[24] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. IEEE International Conference on CAD*, 1993.

[25] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[26] D. Sieling and I. Wegener. Reduction of obdds in linear time. *Information Processing Letters*, 48, 1993.

[27] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3:331–360, 1991.

[28] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, 1994.

[29] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995. Appears also as Duke University Dept. of Computer Science technical report CS-1995-18.

[30] J. S. Vitter. Efficient memory access in large-scale computation (invited paper). In *Symposium on Theoretical Aspects of Computer Science, LNCS 480*, pages 26–41, 1991.

[31] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

## Recent Publications in the BRICS Report Series