Sampling Schemes for Computing OBDD Variable Orderings

Jawahar Jain\textsuperscript{1} William Adams\textsuperscript{2} Masahiro Fujita\textsuperscript{1}

Abstract: We suggest some novel variable ordering techniques based upon the notion of sampling. Such techniques can produce highly effective static variable orders, and can thus be employed in numerous problems where current static variable-ordering techniques prove totally inadequate. They can also augment various reordering techniques thereby helping to produce far superior variable orders in a comparable, or lesser, amount of time. Importantly, we have been able to build BDDs of circuits which could not be represented previously using numerous other reordering packages.

1 Introduction

Ordered Binary Decision Diagrams [3] have found extensive use in various algorithms for analyzing Boolean functions, and thus play an important role in numerous CAD related applications. Unfortunately, in many applications, very large sized BDDs can be generated which can render a BDD based analysis scheme impractical or inefficient. In particular, OBDD sizes are very sensitive to the order chosen on input variables; any “carelessness” in choosing orders can easily lead to exponentially large graphs even though the given function could easily have been represented as a very compact OBDD under a good variable order [2, 9, 21].

In general, for a given \( n \) variable function \( f(x_1, ..., x_n) \), there are as many as \( n! \) different orders on the input variables that can be chosen. In fact, determining the optimal variable order is an NP-complete problem [2]. The best known algorithm to determine the optimal variable order has a complexity of \( O(n^4 \cdot 3^n) \), and is practical only for functions of a very small number of variables [7]. Given the NP-hard complexity of this problem, using heuristics is unavoidable. Coming up with good ordering heuristics, however, has been one of most vexing problems with OBDDs since their advent.

In this context, we note that although many researchers have introduced alternate representations for Boolean functions which may be more compact than OBDDs (sometimes exponentially so) on certain classes of functions, such as gBDDs [1], OFDDs [5], Free BDDs [11], IBDs [16], partitioned OBDDs [17, ?] etc., each of these representations requires use of good variable ordering techniques in one guise or other. Thus, an intelligent selection of a good variable ordering is critical for efficient symbolic manipulation.

1.1 Focus of Our Work

We describe a new variable ordering method which exploits the notion of sampling in the context of dynamic variable ordering. Usually our sampling techniques involve analyzing a portion of Boolean space using reordering techniques, and using the information obtained to help compute the order for the rest of the function. Our results show that:

1. Sampling based dynamic variable ordering can provide very good quality “initial” variable orders which can be used as a static order in numerous real-life function analysis applications. This static order is far more efficient than the current static ordering schemes based on structural analysis such as depth-first search etc. The total time required to compute such static variable orders is only a small fraction of what would be required if the total variable order is computed by analyzing the complete function space using typical variable reordering algorithms/packages [27, 28, 29].

2. The initial, good, static order (sampling-computed order) can be improved when subjected to further analysis. It can be refined to a highly sophisticated order which can be orders of magnitude superior to what is produced by the conventional reordering packages in both space as well as in time. Impressively, we have been able to build BDs of many circuits for which previously the reordering package was not able to construct a monolithic BDD.

To provide a context for our work, we first present the existing state of the art in computing variable orders and their weaknesses in Section 2 and 3 respectively. After providing the intuition behind our approach (Section 4), we present it in detail in Section 5. Our experimental results are presented in Section 6, and an overview of the present work and future research directions is presented in Section 7.

2 State of the Art in Computing Variable Orders

Various ordering heuristics can be classified as either static or dynamic approaches; we will discuss each approach in detail in the following.

2.1 Static Variable Ordering

Static approaches have been very extensively investigated and used for more than a decade [4, 9, 21]. Such heuristics depend on capturing various properties of the given functional description using a structural analysis. For example, using depth-first, or breadth-first, search approaches, “topological closeness” of variables is determined [9, 21]. The variables are now chosen in the manner in which they appear to be most naturally together in the structural description. Lacking adequate functional analysis, such topology based static ordering schemes often fail to appropriately rank (order) variables of a given function. Thus depending on the mode chosen to describe the given function, these heuristics frequently need be changed, or are awkward to use, or are simply ineffective.

2.2 Dynamic Variable Ordering

In a dynamic approach to compute variable ordering [6, 10, 12, 22, 29], one starts with an initial order, which is then analyzed and permuted at internal points in the circuit/function, such that some cost function (often the size of the resulting BDD) is minimized. Dynamic techniques based on simulated annealing [22] or genetic algorithms [6], however, are not very efficient in time. In the dynamic ordering approach of [12], a small set of \( k \) variables are exchanged to determine the
best order on all \( k! \) combinations of these variables. However, the technique has only a limited effectiveness in practically producing good variable orders. In a technique proposed in [18], variable order is calculated by computing ambiguity-reduction in a Boolean function after tentatively fixing the order of a given variable. (The ambiguity-reduction of a function \( f \) by a variable \( v \) is measured by the number of min terms in \( f \) for which the value of the function depends on the value of \( v \).) The variables whose selection reduces the functional ambiguity by most are ordered first [18]. However, such a technique can be also extremely expensive in runtime.

### Sifting-based Approaches to Variable Ordering:

A sifting-based dynamic ordering approach described by [29] appears to be currently the most popular approach to the ordering problem. In this procedure a periodic reordering of variables is attempted to reduce memory requirements. In an OBDD package, each time the graph sizes exceed the original size by a certain proportion, the reordering method can be triggered to reduce the graph sizes. Given a graph \( G \), a variable \( v \) is moved (sifted) to each position in the ordering list and the resulting graph size is examined. The variable is finally assigned the position which results in the smallest graph size. The process is repeated for each variable in the graph. A less expensive procedure may also be used where variables are reordered in a window of say, 3 consecutive variables. This window is then moved forward to include the next variable in the graph, and the process is repeated till all \( n \) variables have been considered. Improvements to sifting-based reordering techniques were suggested by [28] where the number of sift operations was reduced by sifting together the symmetric variable pairs. Further improvements were suggested in [27] where the concept of extended symmetry was introduced to group a larger block of variables.

### 3 Limitations of Current Techniques

The current state of the art variable ordering techniques have limitations in both their time and space performance.

#### 3.1 Dynamic Reordering & Efficiency Issues

This limitation can again be described under two broad categories: inefficiency in time and inefficiency in space.

##### 3.1.1 Time Performance

The dynamic reordering approach can be extremely time-consuming. Reordering is invoked whenever the graph size growth exceeds certain limits. Sifting \( n \) variables, in a graph of size \( G \), has complexity \( O(n \cdot |G|) \). When sifting is called repeatedly, as it happens on hard circuits, the computational costs can be very high. For example, we have a particular circuit (from a controller unit of a parallel processor) for which computing BDDs using a dynamic ordering package took almost 24 hours. However, given that order, the BDDs of the outputs could be computed in only 1 minute! In fact, as the Table 1 shows, such a circuit is clearly not an isolated phenomenon.

### Table 1: Comparison of time (in sec.) spent in reordering vs. building OBDDs for hard circuits from the final order.

<table>
<thead>
<tr>
<th>ckt</th>
<th>time for reordering</th>
<th>time to build final BDD</th>
<th>Final BDD Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>COF(210)</td>
<td>129.26</td>
<td>1.48</td>
<td>3942</td>
</tr>
<tr>
<td>LT3(31)</td>
<td>295.50</td>
<td>1.41</td>
<td>1608</td>
</tr>
<tr>
<td>CR(34)</td>
<td>301.60</td>
<td>1.23</td>
<td>20164</td>
</tr>
<tr>
<td>COF(205)</td>
<td>401.12</td>
<td>37.26</td>
<td>117727</td>
</tr>
<tr>
<td>CR(34)</td>
<td>5311.70</td>
<td>11.56</td>
<td>338931</td>
</tr>
</tbody>
</table>

In practice, these times can be significantly worse.

#### 3.1.2 Space Performance

Sifting-based reordering methods can fail to construct the BDDs for many functions actually tractable for BDDs. This often happens because sifting, a greedy method, can enter a local minimum from which further improvement in variable ordering is very hard. Also, sometimes the orders calculated at the intermediate gates in the circuit are not adequate for the subsequent gates (at any intermediate gate the reordering package is oblivious to the ordering requirements of the functions at the subsequent gates). In absence of this forward-looking capability, a reordering method may fail to find a good variable order for the whole circuit even though one may exist.

#### 3.2 Need for Efficient Static Ordering

Dynamic reordering techniques cannot be used in all applications. There are many applications in which the variable order must not change during the building of a BDD. The lack of good static ordering schemes greatly limits the applicability of some analysis techniques such as probabilistic verification [19], and, we believe, it has greatly impeded research into alternative BDD representations (such as gBDDs [1], IBM DDs [16], XBDDs [19], etc.). Also, in many test-generation applications and SAT-solvers determining some type of order on input variables in which the search must begin is of high importance [30]. For this purpose, we can use ordering techniques similar to ones used for building BDDs of a given function. It is usually not practical, however, to obtain the required order by constructing the complete BDD of the given function using the reordering technique. Again, a capable static ordering technique is essential here. Finally, we note that there are many BDD packages used in industry and academia whose implementation is tailored to local applications, and do not (currently) support reordering techniques. Such packages can also critically benefit if an efficient variable ordering can be provided by interfacing it with some efficient static ordering package.

### 4 Our Sampling Method: Basic Concept

We address the following problem. Given a function \( f: B^n \rightarrow B \), with the set \( I = \{x_1, \ldots, x_n\} \) of input variables, we want to represent \( f \) as an OBDD over the set of input variables. What is an order on \( I \) that allows the BDD for \( f \) to be built efficiently, with a final size close to optimal?

#### 4.1 The Ordering Puzzle

Structural static techniques are usually not effective since they do not take into account much of the functional information. Dynamic methods can calculate local functional information (during BDD building), but lack knowledge of what functions remain to be processed. Intuitively, we will like to base our selection of a variable order by using as much global functional information as possible (note the techniques of [18] (see Section 2.2), though extremely expensive, illustrate such an approach). However, this poses a fundamental problem: how may one obtain such, relevant, functional information?
Obtaining global functional information invariably requires extensive functional manipulation. This, in turn, requires the availability of compact representations that can be manipulated. But compact representations that can be efficiently manipulated seem to invariably require one to have some type of an order fixed on the input variables. So, we are back to square one, in the sense that we may need a good variable order to calculate a good variable order!

4.2 A Solution to the Ordering Puzzle

Since carrying out functional manipulation on the complete Boolean space presupposes availability of the very solution which we are seeking, one way to solve the above conundrum is to carry out the functional manipulation on only some limited subspace. Note, a smaller subspace of a function \( f : B^n \rightarrow B \) can be obtained by restricting the function to a subset \( w \) of all possible input vectors. We use the terminology of of [17], and refer to such a subset (subspace of the given Boolean space) as a window, or alternatively as a window function. Intuitively, a window function \( w : B^n \rightarrow B \) represents a part of the Boolean space over which \( f \) is defined.\(^1\) Conceptually, then, we will like to proceed as in following:

**Our Concept of a Sampling Based Solution:**

(A) Given a function \( f \), we compute, and then analyze various subspaces of \( f \) using a greedy ordering technique. More precisely, we obtain subspaces \( w_1, \ldots, w_i \), and analyze functions \( f_1, \ldots, f_i \), where \( f_i = f \wedge w_i \).

(B) From such reordered-BDDs/variable orders of individual samples, we can compute functional information to deduce the final variable order for the complete Boolean space.

The functions \( f_1, \ldots, f_i \) can be obtained by either an explicit analysis technique, randomly, or by carefully mixing a random selection approach with some form of an explicit analysis. One easy way to obtain such functions (subspaces) is to randomly generate \( t \) cubes, \( c_1, \ldots, c_t \), and obtain a set of restricted functions \( f_{c_1}, \ldots, f_{c_t} \). Of course, a useful sampling based solution must allow the use of functions other than cubes, and the window selection mechanism need not rely on a pure random selection.

By sampling we refer to the technique of examining some randomly or carefully chosen part of a function to derive information (here, a variable order) applicable to the function as a whole. When the subspaces are randomly chosen we can increase its robustness by examining multiple samples, and then either selecting the best from among sample results, or combining the results in some way. Each sample can be regarded as modifying/perturbing the property of the given function. Now, if we run a greedy ordering procedure (for example, sifting-based reordering) on each such function then we may obtain a variety of variable orders. Some of these “perturbed” functions may yield orders which are more efficient, if the sifting procedures are less severely trapped in a local minimum. Likewise, some circuits may yield variable orders which may prove to be very inefficient for the original function. Our method strives to reject the inefficient orders, but pick and further evolve the useful variable orders.

4.3 Some Missing Pieces

To develop effective sampling techniques, the following questions need to be answered:

**Q1. How to sample a Boolean function?**

**Q2. How to determine which samples are likely to be useful in computing an efficient variable order?**

**Q3. How to deduce the final order from information provided by the samples deemed useful?**

We believe if one can work out the above details carefully then one may well have, at least, partially overcome some of the problems associated with the current state of the art. For example, since during sampling we will look at only small graphs, the reordering should run fast. We will also have a forward-looking capability since we deduce the final order for \( f \) only after making the BDDs of samples \( f_1, \ldots, f_t \), a process which requires one to examine the internal gates too. An order computed on a subspace using functional analysis may prove to be a better static order than those provided by existing structural approaches.

5 Our Sampling Method: Details

We will now describe the algorithms used to calculate a good variable order for building the BDD for an output \( F \) of a given circuit \( C \) where node \( F \) has the set \( I = \{ t_1, \ldots, t_n \} \) of input variables in its transitive fanin, and represents a Boolean function \( f : B^n \rightarrow B \). Our ordering algorithm has the following 4 phases:

1) **Sampling for Estimation:** This phase heuristically decides the parameters related to the size of subspaces used for calculating the candidate variable orders by first examining the behavior of the given function on some very small subspaces. Note, if the given function is very difficult to analyze then such behavior may be apparent in even very small subspaces.

2) **Sampling for Candidate Order Determination:** This phase generates a set of candidate variable orders. Essentially, we generate \( t \) number of simpler functions (currently, using randomly generated cubes), and for each function we independently call an ordering algorithm (currently, sifting based reordering procedure). This provides up to \( t \) different candidate variable orders.

3) **Sampling for Testing:** This phase weeds out the ineffective sample orders using an independently generated sample space on which each order generated in the previous phases is used to create BDDs. The candidate orders that perform poorly are rejected.

4) **Sampling for Evolution:** This phase allows for the given candidate sample orders to be further improved, by reordering them independently obtained subspaces, to increase their chance of being suitable for the complete Boolean space. Given \( m \) variable orders as input to this phase, the variable order that gave the smallest graph, on the subspace chosen for evolution, is selected as the sampling-computed variable order.

In the following we will give more details of each of the above phases. Note, the purpose of estimation phase is to get an initial idea about the properties of the given function, so that a more educated guess can be taken about the parameters used to get the final variable order. Also, since the basic sampling techniques used here are similar to what are employed in the subsequent sampling phases, we will further comment on the estimation phase only after giving some more details about the other sampling phases.
5.1 Sampling for Candidate Order Determination

We will first explain the basic sampling mechanism followed by the method for candidate order determination.

5.1.1 Subspace based Sampling

The sampling technique we primarily use is based on subspace analysis. In our implementation as reported in this paper, we obtained subspaces using partial assignments, though the procedure outlined above is not limited to this narrow definition of subspaces.

An assignment function over the set of variables \( J \), where \(|J| = k\), is a conjunction of \( k \) literals, one for each element of \( J \). Thus the assignment function \( a \) can be regarded as a function which assigns true or false to each element of \( J \), depending on whether the corresponding literal is positive or negative in \( a \). Function \( a \) is a partial assignment over the set \( J \), the set of input variables, if it is an assignment over \( J \), for \( J \subseteq I \).

For our given set of variables \( I \), we select a subset \( J \) of these variables and then choose a set \( A \) consisting of \( r \) randomly chosen distinct assignment functions over \( J \). We then consider the set of functions \( S = \{ f_a | a \in A \} \). Here \( f_a \) is the cofactor of function \( f \) on \( a \). We call each function in \( S \) a sample of \( f \).

In our program, we choose \( k \) and \( t \) depending on the size of the circuit being considered, and choose \( J \) to be the first \( k \) variables in the order returned by a depth-first search heuristic applied to output \( F \) - we refer to this order as DFS-order(\( F \)). (A more promising method for selecting the partial assignment variables may be obtained using the techniques discussed in Section 5.6.1.)

We obtain \( t \) variable orders by building BDDs for each of the samples independently. Each BDD is built in a separate BDD manager, with initial variable order DFS-order(\( F \)), in presence of dynamic reordering. For each sample, the program reports the final variable order, and the size of the sample BDD under this order. We prepend the variables from \( J \), in order of DFS-order(\( F \)), to each of these orders.

5.1.2 What are our Candidate Variable Orders?

Each of the \( t \) variable orders produced above can be passed to the testing phase. However, one can easily generate some new candidate variable orders by some simple averaging techniques, which in some cases can lead to a very effective variable order.

Averaging for Determining Candidate Variable Orders: In this approach, we apply some function to the \( t \) sample orders to give an overall order which best satisfies them all, by some measure.

It is by no means obvious what procedure to use to determine such an average order from a given set of orders.\(^2\) Our approach to this question has thus been pragmatic. We defined a number of candidate average functions and tested them to see which performed best in practice.

For \( I \) a finite set of variables, we define an order over \( I \) to be a sequence of elements of \( I \) where no element appears more than once. For \( u \) an order over \( I \), we write \( i \in u \) iff \( i \) appears in \( u \); for \( i \in u \), we write \( pos(i, u) \) for the position (index) of \( i \) in \( u \).

Suppose for some \( f \) and for all \( r \), \( 0 \leq r < t \), we are given an order \( u_r \) and a positive integer-valued weight \( w_r \), where for each \( i \in I \), there is an \( r \) such that \( i \in u_r \), we define the rank of each element of \( I \) as follows (in each of the sums, \( r \) ranges over values such that \( i \in u_r \)).

\[
rank(i) = \frac{\sum_r \{pos(i, u_r) \times w_r\}}{\sum_r w_r}
\]

The average order of the inputs is then defined as the indices in \( I \) in increasing order of rank.

We experimented with a number of averaging functions, using different choices for the weights \( w_r \). One choice we used was the size of the sample BDD under its final order. The intuition for this choice is that if, despite reordering, some sample gives a large BDD, then we should give a greater consideration to the order found for this sample in the final average than to the order for a sample function that has a small reordered size. We found this to be the most successful averaging method of those we tried. We will like to investigate in future techniques that can merge different variable orders to produce a single order such as in [8].

5.2 Sampling for Testing

In this phase, we take each of the sample orders and test them by applying them to a fresh set of samples of \( f \). The order which gives the smallest total size (taking account of BDD node sharing) is chosen as the best order. In general we find that the testing phase is quite successful in weeding out the "bad" variable orders. If in the preceding phases it is guessed that the given function may have large BDDs, then during the testing phase we allow dynamic reordering during the building of the BDDs for the samples for hard function. Further, we use explicit reordering to reduce the final size of the set of BDDs built. In such cases, the sample order is changed during the testing. In effect, we use the testing samples to refine the sample orders. This refinement, only an implicit aim of this phase, is explicitly targeted in the next sampling phase, the evolution phase.

5.3 Sampling for Evolution

Given a set of variable orders, this phase tries to determine which given variable order adjusts better (allows meaningful evolution) to newly chosen subspaces, i.e., which order is less severely stuck in a local minimum. It also attempts to weed out the less effective of the variable orders that could not be filtered by the testing phase. Conceptually one may repeat the testing phase and the evolution phase multiple times.

In our implementation we use similar techniques for the sampling for testing phase, and sampling for evolution phase. In testing phase, we test (and refine) the sample orders against a small number of randomly chosen samples using a large number of assignment variables. (This generally gives small sample BDDs which can be rapidly built since many variables have been removed.) In the evolution phase, we choose the best order(s) produced from the testing phase, and refine them against another set of randomly chosen samples, this time with a smaller number of assignment variables. We find that a use of evolution phase is not required for the easy functions. However, for hard functions, using evolution phase reduces the variations in the quality of the variable orders produced, as well as produces smaller output graphs.
Note, the parameters selected in all of the phases discussed can either be supplied by the user (thus allowing a welcome flexibility in controlling the quality/cost of reordering solutions), or be decided with the help of an estimation phase.

5.4 Estimation Phase

Two key issues that remain to be discussed are the number of variables to be employed in each partial assignment, and the number of partial assignments (samples) to use. We make an initial appraisal of the number of sampling variables, as well as the number of samples used for the candidate order determination phase, as well as the subsequent phases, after a quick analysis of the given circuits for its level of difficulty. The basic principle of our estimation method is to examine the size of BDDs that result if we make relatively large number of rather small sized samples. The large number of samples decreases the effect of randomness; the smaller sample sizes make the estimation phase practical. Then, we correlate the sample BDD sizes with the size of the circuit. Although such correlation is only heuristic, it can help decide on how many variables to consider for partitioning the Boolean space. The estimation phase can be repeated with a lesser number of variables if the original estimation, using very small samples, results in graphs too small to calculate meaningful information.

5.5 Sampling Parameters

In our implementation, depending on the size of circuits, sampling variables and number of samples selected varies. One may obtain reasonable results if the number of variables used in the candidate order selection phase varies from 3 to 10 (with 1 to 5 samples), number of variables used in the testing phase varies from 5 to 15 (with 1 to 4 samples), and in the evolution phase, varies from 3 to 8 variables (with 1 to 5 samples). In our program, the number of samples were varied with the size of the circuit, and for smaller circuits, a lesser number of samples were found to suffice. Similarly, if the size of the BDD becomes too large then we use a larger number of variables during sampling. Also, if a trivially sized BDD is generated then that sample is dropped from the sample set under consideration. Note, the decision whether the program enters the testing phase and the evolution phase is based upon the characteristics (sizes) of graphs generated in the estimation and candidate-order phases.

Also, note, many circuits could contain a large number of easy output functions with a rather small support set. Hence, in applying our technique to find an order for building BDDs for all the outputs of a circuit simultaneously, we found it better to consider a reduced circuit. Our algorithm considers output nodes in decreasing order of size of their input cone, and terminates, returning a set of output nodes, once it has found enough outputs that some given proportion of the original input nodes (which choose 95%) will be left in the reduced circuit.

Interestingly, we find that quality of results may be improved by selectively rejecting some samples if their partial assignments violate some predefined criteria. Type of circuit produced after simulating a given partial assignment, or similarity between any partial assignment generated with the set of previously selected partial assignments are two of such selection criteria.

Further details of our method are subject to a pending patent application [13].

5.6 Other Supporting Sampling Techniques

In the following we describe some other sampling techniques that augment the basic framework described above.

5.6.1 Circuit Sampling: Limited Circuit Exploration

Apart from subspace based sampling, another method of analyzing a limited portion of the given function is the circuit sampling technique. In such a technique we run reordering methods for either a fixed number of levels in the circuit or till number of reordering calls have exceeded some threshold for the given circuit. Circuit sampling techniques can be used in conjunction with subspace-based sampling. An interesting use of circuit sampling that we have already explored is to provide the initial order from which the subspace-based sampling must start. When carefully used, we find it to be a very effective technique to provide the initial order to standard reordering methods. Our initial experimentation has yielded useful results, thus we believe its use should be explored further in the current state of the art.

5.6.2 Sampling By Limiting Manager Size

A recent publication at IWLS98 [23] also uses the term “sampling” in the context of improving sifting based variable ordering. To save on reorder time, their techniques can limit the size of the BDD being reordered to a fraction (say, 25%) of the existing nodes in the BDD manager; this fraction can be varied if the results obtained are not satisfactory. Such a sampling technique has its merits but it does not enjoy the forward-looking capability that subspace analysis provides, as well as estimation and testing/evolution phases which are critical to reduce variance in the results of random sampling on Boolean functions. In the future we plan to study in detail a possible integration of our techniques with those in [23].

6 Results

In the following we will report how the use of sampling techniques can generate an initial variable order (sampling-computed order) that when supplied to reordering techniques can significantly improve their performance. We discovered that some functions which are extremely difficult for OBDDs can be consistently processed using sampling based methods. We also did not find any case where a function can be processed with the CUDD package, but where the sampling methods proved inadequate. We will also see how the sampling computed order can consistently prove to be an effective static variable order even though it was obtained without analysis of the complete Boolean space.

Experimental Setup: For the experimental runs, our algorithms were coded in C++, in SIS environment, and used Colorado University Decision Diagram Package (CUDD-2.1.2) [31]. The program was run on Sun Sparc-Ultra30 desktop machines with 512MB RAM, except for the experiments described in Table 4 where we used a dedicated SUN Sparc-20 compute-server with 512MB RAM, and 2GB swap space.

The functions used for benchmarking: The test circuits used represent a wide variety of industrial designs of Fujitsu; the OBDD packages fail on many of these circuits. Hence such circuits serve as important test cases for efficient BDD algorithms including the variable ordering algorithms. We demonstrate our results on many of such hard industrial
circuits. However, since these circuits are not available in the public domain, we also show our results on the known hard output functions from ISCAS-85 benchmarks. Although our method has been effectively applied for building BDDs for multiple output cases also, in this paper we only focus on the analysis of hard, single-output, Boolean functions. In fact, if a given output in an intractable circuit is proving to be very hard to represent, it becomes even more unlikely to be able to succeed in representing such a circuit with all outputs analyzed together. Also, in this context, we note that for many Boolean analysis problems, we have shown by extensive experiments that analysis of difficult functions can be automatically broken into an analysis that is conducted on an output by output basis.\footnote{To achieve robust as well as efficient solution to the verification problem, a filter based mechanism was developed in [4, 24]. The filter approach is a combination of communicating techniques where each technique calculates (filters out) the information it is most suited for, alters the circuit accordingly, and passes (passes) its results to the subsequent techniques (filters). Typically, easier cases of verification are handled first with fast, low cost filters, followed by more complex and expensive filters that have a higher time and space complexity. Using the reasoning mechanisms provided by the filter theory, various orthogonal verification approaches such as OBDDs, SAT/ATPG, learning, partitioning, BDD-hash, structural approaches can be systematically integrated to create a working prototype of an extremely efficient verification procedure.}

The above analysis holds for each of the hard industrial circuit that we have considered. Thus we chose to report results by considering only the hard functions from the ISCAS-85 as well as some Fujitsu circuits. For the ISCAS-85 circuits, we find following output functions to be good test cases for variable ordering algorithms: c432: 432gat, c499: od31, c1355: 1355gat, c1908: 72, c3540: 402. Apart from this, for c6288.11, we chose the 11th output (494gat), and for c6288.12, the 12th output, 530gat was chosen.

**Description of Tables:** In our tables, the “DFS” entry refers to the DFS based static variable ordering method implemented in SIS. Similarly, the CUDD entry refers to the CUD-2.1.1 (using sift). “By Sampling” refers to the sampling technique discussed in Section 5. Both CUDD and the sampling method were provided the SIS DFS order as the initial variable order. Generated over 40 runs, we report the median, mean, standard deviation, and range of the output graph sizes as well as the median and the mean runtimes. Runtime entries include both the time taken in the sampling phases and the total runtime. (Note, the total time includes the time taken for the sampling phases, as well as the time taken to construct the final BDD from the sampling-computed order). In our tables, all statistical data about BDD size is measured in number of BDD nodes, and all run-times are reported in seconds.

**6.1 Table 2: Sampling as a Static Ordering Method**

Using the sampling methods the current state of the art in computing static variable ordering can be vastly improved (see Table 2). Using the static order provided by the sampling method we are now able to process a much larger set of the hard functions that cannot be processed using standard DFS order. The run-times taken by the sampling method can be controlled depending upon the amount of sampling carried out. We find that as we decrease the amount of sampling, the runtime also decreases proportionately, along with an increase in the size of the graph produced. For example, in CRC24, if we decrease the amount of sampling then though our median graph size increased to 377012, our median sampling time decreased to 20.87 seconds! Similarly, for COP210, with the median sampling time of 10 seconds, we obtained a graph of 383377 nodes, still much smaller than 1.72 Million nodes obtained by the DFS techniques. However, the more significant point is that without analyzing the full Boolean space we could calculate variable orders that are often as good as those calculated by dynamic ordering.

**6.2 Table 3 and Table 4: Advantage of Sampling with Sifting**

In the following we will report how the efficacy of the (standard) sift-based ordering routines, as implemented in the CUDD package, can be dramatically improved with the use of our techniques. CUDD provides a range of reordering methods such as sift, sift-converge, symmetric-sift, group-sift, etc. Note sift appears to be the most common reordering method that is employed in many other public as well as proprietary reordering packages. Thus, the enhancement results we report are on the sift reordering method. That is, in our tables we compare the sift reordering and the sift augmented with sampling. (We ran many experiments using group-sift, and found similar results, so we do not believe the results are a consequence of the choice of reordering method.)

On harder functions we find that the use of sampling can make sifting based methods far more powerful. For some circuits where sifting cannot build BDDs of the given circuit (see Table 4), augmented-sifting with sampling can do the job even though the total fraction of Boolean space sampled over all 4 sampling phases was less than 10%. For circuits which are deemed less difficult, sampling a lesser fraction of Boolean space usually suffices. The fraction of Boolean space that is required to be sampled however can dramatically change for circuits whose sample BDDs under initial order have a very peculiar structure (a large number of short paths or a highly shared graph structure are two of such examples).

Usually we find that through a careful use of sampling, a good initial order can be found relatively quickly. Use of a good initial order can often reduce the total number of reordering calls. Thus, the resulting method may be more time-efficient even though the final graph sizes are similar.

One explanation of superior performance of augmented-sifting using sampling is that the sampling-computed order provides a starting point from which it is easier to reach the solution space of good variable orders. This seems to be especially true when for a given circuit, some forward-looking capability must be present in an ordering algorithm.

**Observations on Sampling for Evolution Phase:**

We have noticed that in many cases, the results of sampling based methods improve dramatically due to the contribution of the sampling for evolution phase. For many functions we obtain variable orders which give graph sizes that are an order of magnitude smaller than what is obtained if we would use the CUDD package augmented only with the earlier phases. For example, when using sampling without the evolution phases, our technique gave an order under which the BDD had 700K nodes for one proprietary circuit (whereas the order from the CUDD package gave a BDD of 300K nodes). However, with the proper use of the evolution phase, our graph sizes dropped to around 6800 nodes! Also, we find that the use of evolution
phase makes the method relatively less dependent on any randomness in the partial assignments chosen on first k variables in the DFS order. In some other circuits, even though the smallest graph size that can be obtained remains unchanged, using the evolution phase dramatically reduces the variance in the resulting graph sizes.

6.3 Weakness of reported implementation

We have recently implemented various improvements to the techniques published in the current paper by using more powerful sampling schemes as compared to the cube based sampling resorted in our current implementation. However, such improvements are still in the process of being established through extensive experiments. The algorithm reported in this paper is not consistently able to filter out inefficient variable orders. This results in large deviations for some circuits due to random fluctuations, and in a limited number of cases, in a rather large-sized final BDD. Thus, although, the average BDD size can be significantly improved by ordering through sampling, we believe the testing/evolution phase may require further research. We believe a more comprehensive examination of the interaction of various sampling phases is also required. Using only partial assignments on input variables to do the sampling is limiting; we are extensively examining the use of functions other than cubes to do the sampling.

7 Summary and Future Work

We have described many effective sampling based bivariate ordering techniques. These techniques fill the void between the conventional static and dynamic variable ordering techniques, and can help improve the state of the art for both. Our results show that our sampling-computed order can provide an initial order starting from which a greedy optimizing technique such as the sifting procedures are more likely to obtain more efficient solutions. The CUD D sift reordering, for example, when augmented with our techniques can produce BDDs for circuits which previously could not be processed. Also, our techniques can provide a good static order without analyzing the full Boolean space using which we can process functions that are intractable for the classical static DFS ordering techniques.

Our selection and evolution techniques can be used with other optimization techniques such as those provided by genetic techniques or simulated annealing. We also plan to study its applicability in the context of reachability analysis/model checking procedures. We are also developing applications of our efficient static ordering technique in other function analysis problems. Another application of our work is to identify functions for which partitioned-OBDDs [17, 26] should be constructed instead of conventional monolithic-OBDDs. When we find that different samples have highly different variable orders then such functions are good candidates for analyzing through partitioned-OBDDs, which we have verified experimentally.

Most important of all, we believe this paper is only an initial demonstration of how good variable orders can be computed using sampling. Thus, we believe further research should easily strengthen the basic techniques that have been employed here. In fact, we also are developing new sampling techniques which have shown potential for even more improved results. Also, our current research indicates that sampling techniques can be usefully employed in many other CAD applications as well [25]. Thus, we believe that these techniques can offer a rich set of possibilities for improving the state of the art in numerous problems of interest.

References

Table 3: Detailed statistical data on some hard Boolean functions to prove that augmenting reordering techniques with sampling can prove helpful.

<table>
<thead>
<tr>
<th>Ckt</th>
<th>SPACE</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CUDD</td>
<td>Median</td>
</tr>
<tr>
<td></td>
<td>Median</td>
<td>Mean</td>
</tr>
<tr>
<td>U432</td>
<td>377</td>
<td>5890</td>
</tr>
<tr>
<td>U409</td>
<td>2100</td>
<td>2100</td>
</tr>
<tr>
<td>U1535</td>
<td>244</td>
<td>2622</td>
</tr>
<tr>
<td>U3908</td>
<td>990</td>
<td>990</td>
</tr>
<tr>
<td>U3540</td>
<td>7883</td>
<td>6507</td>
</tr>
<tr>
<td>UG288(11)</td>
<td>20530</td>
<td>13880</td>
</tr>
<tr>
<td>UG288(12)</td>
<td>40666</td>
<td>33040</td>
</tr>
<tr>
<td>HRC(24)</td>
<td>26164</td>
<td>19311</td>
</tr>
<tr>
<td>COP(110)</td>
<td>3841</td>
<td>3885</td>
</tr>
<tr>
<td>COP(21)</td>
<td>1408</td>
<td>1408</td>
</tr>
<tr>
<td>COP(21)</td>
<td>2000</td>
<td>1408</td>
</tr>
<tr>
<td>HRC(54)</td>
<td>3393</td>
<td>35815</td>
</tr>
</tbody>
</table>

Table 4: A selected group of some hard industrial circuits. Data is reported over 7 runs for last 3 (hard) entries, and over 20 runs for COP(205). “fail” for CUDD indicates cases where it was run from 50 to 80 hours before being aborted.

<table>
<thead>
<tr>
<th>Ckt</th>
<th>SPACE</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DFS</td>
<td>CUDD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Median</td>
</tr>
<tr>
<td>COP(205)</td>
<td>fail</td>
<td>117272</td>
</tr>
<tr>
<td>COP(210)</td>
<td>fail</td>
<td>258892</td>
</tr>
<tr>
<td>COP(200)</td>
<td>fail</td>
<td>177964</td>
</tr>
<tr>
<td>FP16</td>
<td>fail</td>
<td>296854</td>
</tr>
</tbody>
</table>