A BDD Variable Reordering Heuristic Based on Output Probability Periodicity

A BDD Variable Reordering Heuristic Based On Output Probability Periodicity

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Systems Engineering

By

Joshua Paul Williams, B.S.C.S.E.
University of Arkansas, 1997

December 1998
University of Arkansas

This thesis is approved for
recommendation to the
Graduate Council

Thesis Director

_____
Dr. Mitchell Thornton

Thesis Committee:

_____
Dr. David Andrews

_____
Dr. David Wessels

THESIS DUPLICATION RELEASE

I hereby authorize the University of Arkansas Libraries to duplicate this thesis when needed for research and/or scholarship.

Agreed_____

ACKNOWLEDGEMENTS

First, I would like to thank Dr. Thornton for serving as my thesis advisor, for providing a challenging project to work on, and lending invaluable advice. I also thank Dr. Andrews and Dr. Wessels for their service on my thesis committee. Finally, I thank the National Science Foundation for sponsoring this project.

Table of Contents

Chapter 1: Introduction

This thesis reports the results of an investigation of using and minimizing a particular type of Boolean function representation.

1.1 Introduction

Digital system design tools include techniques for logic synthesis, verification and test vector generation, among others. These tools all require the ability to internally represent Boolean equations. Solutions to these design problems generally require manipulation of the Boolean functions.

Digital CAD tools must use the most efficient data structures in representing the Boolean functions. Among the data structures that have been used to represent Boolean functions are truth tables and cube sets. A truth table is a table that lists every possible combination of the inputs for a given function. The truth table also indicates which combinations cause the function to evaluate to a "logical one". A cube set is a visual representation of the function. For each input, an extra dimension is added to the cube set. Each vertex in

the cube set represents a combination of inputs. By joining adjacent vertices whose input sequence causes the function to equate to a logical one, the size of the cube set can be reduced.

Using these data structures, however, has proven to be difficult due to their large size. Consider representing a function that has n inputs with a truth table. For this function, there are 2^n rows in the truth table that are needed to accurately represent the function. It is not unusual to have functions that have many inputs; functions with 40 inputs are quite plausible and would require 2^{40} rows for a truth table representation. Cube sets can reduce the memory requirements by at least one half. Cube sets also do not necessarily provide unique representations and reducing the size of a cube set can be time consuming.

One solution for efficiently representing Boolean function involves the use of Binary Decision Diagrams. The Binary Decision Diagram (BDD) is a graph based data structure that provides a behavioral representation of a Boolean function. More specifically, it is a directed acyclic graph (DAG), similar to a binary tree in appearance though it does not adhere to the same restrictions that binary trees do. For example, a binary tree allows for only one incoming edge for any given node. In a BDD, multiple edges may point to the same node. Usage of the BDD data structure has proven to be an effective representation for many functions [4]. However, the BDD approach to representing Boolean equations is not without its flaws. In some instances, BDD representations can grow to exponential sizes with respect to the number of inputs. Therefore, it is necessary to develop size

reducing techniques for BDDs so that they provide the most compact representation for Boolean functions. Such techniques should use as little memory as possible and reduce the size of the BDD representation as quickly as possible.

1.2 Binary Decision Diagrams

This section will provide a definition and a brief history of the BDD data structure. Descriptions of some of the variations that can be applied to the BDD data structure are included.

1.2.1 Binary Decision Diagrams Defined

Binary Decision Diagrams were introduced in 1978 when Akers developed what is known as the Free Form Binary Decision Diagram [1]. This development was derived from work done in the 1950's by Lee [2], which involved using a technique known as Binary-Decision Programs to represent switching circuits. The BDD data structure formed by Akers is a behavioral, graphical representation of a set of Boolean functions. Akers's BDDs have two types of vertices, or graph nodes. The first type of node is the non-terminal node. Non-terminals nodes have three attributes associated with them: an index, a 1-edge, and a 0-edge. The index is used to represent a particular variable (input) of the function that the BDD represents. The edges are traversed based on the value of the index of the node. For example, if a node index takes on a value of 0, then the 0-edge would be followed to the node that it points to; should the index value be 1, then the 1-

edge would be traversed. The terminal nodes have a single attribute, which is a Boolean constant of either 1 or 0.

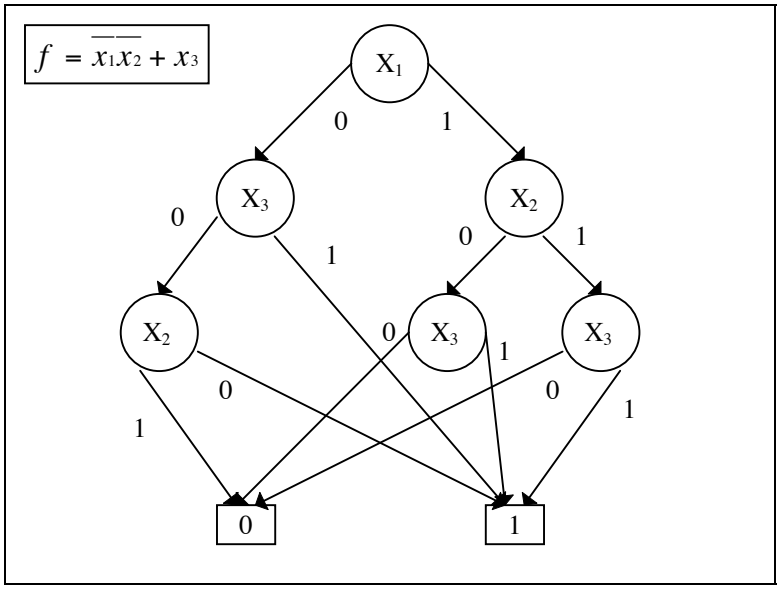

Figure 1-1: An Example BDD

The example BDD in Figure 1-1 illustrates how a BDD defined by [1] might be used to represent the indicated function. A BDD is traversed by starting at the root node, which has the index x_1 in this example. Suppose that x_1 takes on a value of 0, then the 0-edge would be traversed down to the next node, which has an index of x_2. The value of this node would then determine the next edge that is traversed, and so on until a terminal node is reached. The terminal node then indicates whether the function evaluated to a logical 0 or to a logical 1. As already stated, BDDs are behavioral in that they indicate which sequence of input values will cause the function to evaluate to either a 0 or a 1.

The definition of a BDD as given in [1] does not always provide canonical representations of Boolean functions. Consider the BDD example in Figure 1-2. It represents the same function as Figure 1-1, but has a more compact form (which helps demonstrate the importance of size-reducing techniques for BDDs). Another problem is that there is no standard for the placement of variables in the BDD. For example, in a BDD defined by Akers, it would be permissible to encounter variables in different orders when traversing alternate paths along the graph. In fact, variables might even be encountered more than once along any given path. Because more than one BDD can represent the same function, using these types of BDDs for tasks such as verification can be difficult.

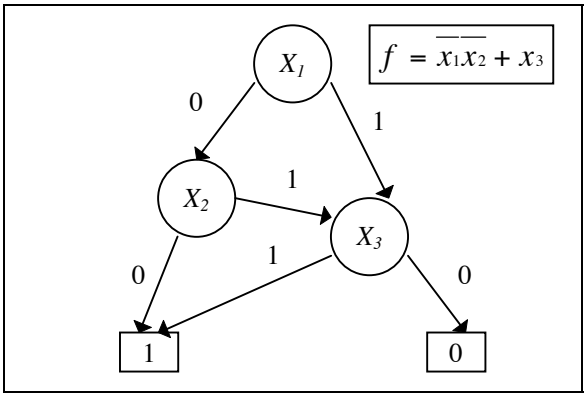

Figure 1-2: Alternate BDD for same function

1.2.2 Reduced Ordered Binary Decision Diagrams

Though Akers creates a useful data structure with the BDD, it needs refinement so that it is more efficient in size. This refinement was provided by Bryant in 1986 [3] when he

developed two restrictions and two reduction rules to be applied to BDDs as defined in [1]. The result was the Reduced Ordered Binary Decision Diagram (ROBDD). The first restriction made was that variables had to be encountered in the same order along any given path. The second restriction was that variables could occur only once along any given path. The first reduction rule involves the removal of redundant nodes. If both a node's 0-edge and 1-edge point to the same node, it is called redundant. Any edges that pointed to a redundant node would be redirected to point at the node referenced by the redundant node. The second reduction rule is to share isomorphic sub-graphs. If a particular sub-graph (a tree of nodes) appears more than once in a BDD structure, then the duplicate copies can be removed; the edges that pointed to the deleted copies, of course, would be redirected to point to the remaining sub-graph.

With these restrictions and reduction rules, BDDs have canonical representations of functions for a given variable order. Also, creating and applying algorithms to manipulate BDDs is easier using the modifications in [3]. Thirdly, the spatial requirements for ROBDDs is generally smaller and ranges from n to 2^n nodes, where n is the number of primary inputs for the function that is represented. In the remainder of this work, the usage of the term BDD will be a reference to the ROBDD as defined in [3].

1.2.3 Negative Edge Attributes

Another technique used to further reduce the spatial requirement of a BDD involves the use of attributed edges. An attributed edge is an indication that some operation is to be

performed on the sub-graph referenced by that edge. In this work, emphasis will be placed on the use of the negative edge attribute [5].

Essentially, the negative-edge attribute complements the referenced sub-graph and can be thought of as an inverter. The negative-edge attribute is advantageous in that it can reduce the size of a BDD up to one-half and has constant time negation. Though helpful, negative-edged attributes must be regulated to prevent non-canonical representations from occurring. One way to ensure that canonical representations are provided, thereby preserving uniqueness, is to issue two stipulations. The first stipulation is that only the terminal node for a logical 1 is used. Second, negative edge attributes can be applied only to 0-edges within the BDD data structure. Uniqueness can also be maintained if a "0" terminal is used and negative attributes are applied to the 1-edges. Regardless of which variation is used, consistency must be maintained if uniqueness is to be preserved [6]. In this work, the former variation is used.

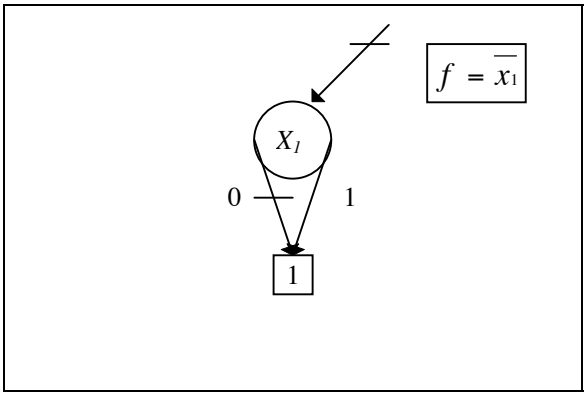

Figure 1-3: A BDD with Negative-Edge Attributes

Figure 1-3 illustrates how negative-edge attributes can be used with a BDD data structure. Note that only the "1" terminal is being used and that a negative attribute appears on the 0-edge. Also note that a negative edge can be applied to the root node's incoming pointer. This has the effect of complementing the entire BDD structure. This is the only exception to where a negative attribute can be placed if uniqueness is to be preserved.

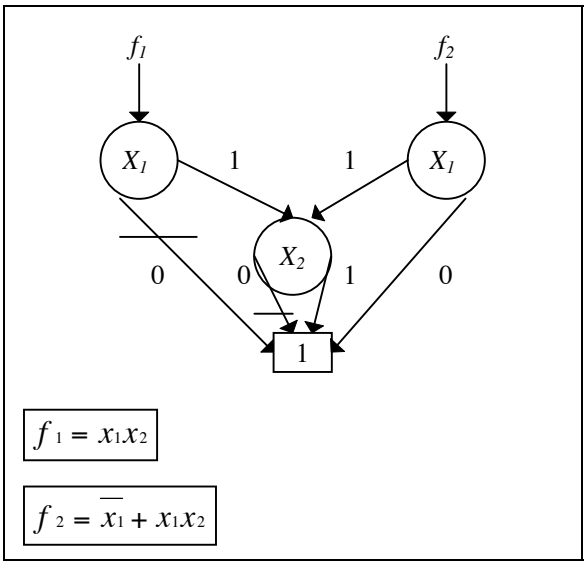

Figure 1-4: A SBDD Example

1.2.4 Shared Binary Decision Diagrams

Often, there are Boolean functions with multiple outputs that are dependent on the same inputs. For example, the sum and carry outputs of an adder function are dependent on the same inputs. It would be inefficient to use a BDD to represent each separate output function when a single BDD could be used instead. This is the principle behind the Shared Binary Decision Diagram (SBDD) [5]. The SBDD is a multi-root graph that is

used to represent multiple functions that share similar logic. An example is provided in Figure 1-4. Note that this example also uses negative edge attributes to further maximize the spatial optimization.

1.3 Survey of Applications

This section explains the value of BDDs in digital design. The use of BDDs in the applications of verification and logic synthesis will be examined.

1.3.1 Verification

First, consider using BDDs to perform formal verification. Formal verification is the process in which it is proven that a property holds of a model of a design [16]. More specifically, formal verification is the manner in which a function is proven to be correct. BDDs can be used to perform formal verification in a number of different ways. One method is checking the combinational equivalence of two circuits. Recall that BDDs can provide canonical representations of Boolean functions; in other words, they provide unique representations. Since a circuit can be represented as a Boolean function, it can also be represented as a BDD. To verify whether two functions are equivalent, a BDD can be formed for each function. If the BDDs are identical, then the two functions are equivalent. BDDs also represent a set of truth assignments for a state machine. Suppose that a state machine has been created, but its functionality is unknown. By indicating which combination of input assignments evaluate to a logical one, the BDD can verify the behavior and function of the state machine. Thus, BDDs can also be used to indicate whether two state machines of different design have the same function. Rather than just

comparing the equivalence of two circuits or verifying the behavior of a circuit, a BDD can also be used for model checking. Model checking is similar to verifying the function of a state machine. The difference is that model checking ensures that a circuit follows specified properties over the course of time. For example, one BDD can be used to represent the current state of a machine while another is used to represent a previous or forthcoming state. Thus, BDDs can determine whether a machine behaves as it should as the values of its inputs change over time [16][17].

Using BDDs for formal verification is not a trivial issue. Formal verification is a very important process in digital design. It is a troubleshooting process by which bugs and errors are removed from a design. Though important, verification is also a time consuming process. Usage of the BDD is one way to speed up the process of verification. There is, however, a problem with using the BDD data structure for verification. For larger, more complicated circuits, BDDs can also become larger and more complicated and more time consuming to use. It is, therefore, important that BDDs be reduced to the smallest possible size. Improvements in the use of BDDs translates to improvements in the use of other applications, such as formal verification. Thus, the development of size-reducing techniques to be applied to BDDs is important and justified.

1.3.2 Logic Synthesis

BDDs may also be used to perform logic synthesis. Logic synthesis is a design process used to optimize the design of a circuit. There are various approaches to using BDDs to perform logic synthesis. For example, suppose a logic designer wishes to check whether

a redundant input has been incorporated into a circuit design. The circuit can be initially represented as a BDD. Then, by removing redundant nodes from the BDD as described in [3], a designer can determine which inputs in the circuit are redundant. Thus, by applying reduction techniques to a BDD representing a particular digital design, the number of states in that design can be reduced [18]. BDDs can also be used to efficiently calculate spectral coefficients for Boolean functions. Spectral coefficients provide information describing the relationship between the inputs and the outputs of a function. In the past, calculating the values of spectral coefficients involved the calculation of inner products of vectors that were exponential in length. With the use of BDDs, spectral coefficients can be calculated in polynomial time [19], thereby reducing the time involved in developing circuit optimizations.

Logic synthesis, like formal verification, is an important process. Creating a structural representation from a behavioral description can be a time consuming and the results may not always be optimal. Since BDDs can be used for logic synthesis, techniques that optimize BDDs can be thought of, indirectly, as techniques that optimize logic synthesis.

1.4 Variable Ordering vs. Variable Reordering

As defined in [3], variables in a given ordering must be encountered at most once along any given path of a BDD. However, if the ordering of the inputs is altered, then the layout and size of the BDD is also altered. Determining the best variable ordering, that is determining the ordering that will reduce a BDD to its optimal (smallest) size, has been proved to be an NP-Complete problem [7]. Since there is no known algorithm that can

solve for the best known ordering in polynomial time, the use of heuristics to determine variable orderings is appropriate. A good heuristic can be defined as one that either produces an ordering that is optimal or an ordering that is close to optimal. Orderings that are close to optimal are called "good" orderings.

There are two versions of the problem of producing a compact BDD. The first of these two methods is called variable ordering. In this approach, a variable ordering is determined prior to the creation of a data structure. Often, a function is subjected to an analysis that will determine the variable ordering used to create the structure. Netlists and other circuit descriptions are often used as the basis for such analyses.

The second version is called variable reordering. Unlike variable ordering, variable reordering requires that an initial data structure, such as a BDD, is created. The BDD is then analyzed to determine a rearrangement of the variables, which results in the reformation of the BDD. The focus of this work is on the development of variable reordering heuristics.

1.4.1 Variable Reordering Techniques

Variable reordering is typically done by exchanging some or all of the inputs in the ordering. A well-known method of exchanging variables is the sifting technique, developed by Rudell [10]. For any Boolean function with n primary inputs, there are $n!$ possible orderings that can be used. The sifting algorithm searches an n^2 sub-space of the possible $n!$ orderings and returns the ordering that resulted in the smallest BDD size. The

results of sifting are exceptional if a “good” sub-space is examined. Otherwise, sifting can be a time consuming procedure and may return “poor” orderings.

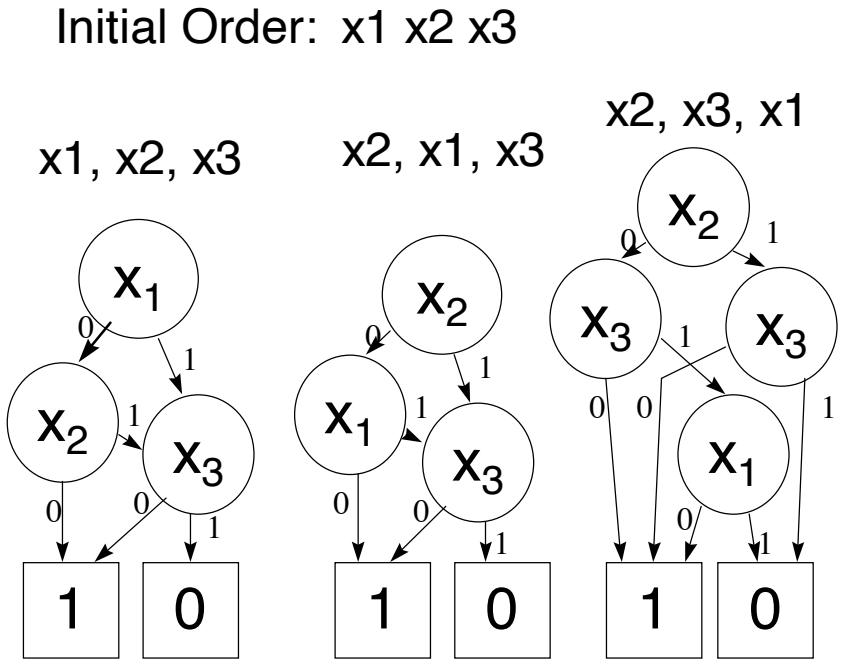

Figure 1-5: Sifting Example

Figure 1-5 gives an example of the sifting process. Consider a function that depends on three inputs, x1, x2, and x3. To perform the sifting process, a variable is selected and exchanged with all of the other variables. That is, the variable is placed into every possible position in the order of inputs. In figure 1-5, the variable x1 is selected and placed in every possible position in the list of input orders. The ordering that produces the smallest BDD is returned by the sifting function. In this example, either the ordering x1 x2 x3 or x2 x1 x3 may be returned since they both produce a BDD of the smallest

possible size; hence, they are "good" orderings. The ordering of x2 x3 x1, however, is a "poor" ordering since it returns a BDD that is larger in size than the other two orderings.

Another reordering technique is Simulated Annealing [11]. In Simulated Annealing, the input order is altered at random and is continually altered (according to a "cooling schedule") until there ceases to be improvements in size. This technique is excellent at creating compact BDDs and has obtained some of the best known sizes for some benchmark circuits. However, the Simulated Annealing process is very slow and is not practical to use in a CAD tool. Also, the results returned by Simulated Annealing are not always the same since it randomly switches the input order [11].

Previous work presented in [8] provides a reordering technique that uses probability based metrics. Here, the probability of the consensus function returning a logical 1 is used to determine reordering metrics. The consensus function is the intersection of the negative and positive cofactors about an input, x_i. A negative cofactor taken about x_i indicates that the variable x_i will take on a logical value of 0 for a given function. The negative cofactor can be represented as $f_{\overline{xi}}$. The positive cofactor indicates that x_i will take on a logical value of 1 and is represented as f_{xi}. The consensus function can then be defined as $C_{xi}(f) = f_{xi} \bullet f_{\overline{xi}}$. The consensus function is useful as an indicator of how much a function's outputs depend on a particular input. This approach works well as a pre-sifting filter for SBDDs with negative-edged attributes.

A minimization technique involving Evolutionary Algorithms (EAs) has also been developed. In the EA approach, two operations are continually applied to an ordering of inputs until there is no further improvement in the ordering. These operations are the sift function, as described above, and the inverse function, which simply reverses the entire input order or a subset of the order. The EA approach works well if a relatively small number of inputs is being reordering. The EA method has obtained some of the best known orderings for several benchmark circuits [15]. Like Simulated Annealing, the EA approach also tends to be slow.

1.5 Contribution of this Work

In this work, a new heuristic for reordering input variables to quickly minimize BDD sizes is developed. This heuristic uses reordering metrics based on the calculation of output probabilities. These probabilities are then used to assign ordering metrics to the inputs. Initially, variables that were symmetric (defined later) were placed close to one another in the order list of inputs. However, it was discovered that placing symmetric variables close to one another did not always yield the best results. After further analysis, it was determined that spacing symmetric variables as far apart as possible could provide better results. This became the main focus of this work and an algorithm that exploits symmetry is used as the foundation of the reordering technique. In addition, a new modification to the sifting technique has also been developed.

1.6 Outline

Chapter 2 provides the mathematical background for this new heuristic. Output probabilities are defined and an algorithm used to calculate these probabilities is described; these probabilities are instrumental in the assignment of ordering metrics. An analysis of the complexity of this algorithm is included. Also described are the characteristics of "good" ordered BDDs. Next, the relationship between "good" orders and probability values is explained. Then, an algorithm that uses symmetry to create "good" orders for BDDs is demonstrated.

In Chapter 3, the heuristic used to reorder BDD inputs is presented. First, a quick review of how symmetry is used to create an order list is provided and the relation between symmetry and probability metrics is explained. Then, the actual technique used to reorder inputs for a BDD is described in full detail. A modification to Rudell's sifting algorithm is also presented. Then, a in-depth example of the heuristic is provided using a benchmark circuit. Experimental results for the heuristic and for the modifications to the sifting algorithm are included and analyzed.

Chapter 4 provides the conclusions of this work. The significance of this heuristic is discussed and future improvements for the heuristic are presented.

Chapter 2: Mathematical Background

This chapter defines an output probability algorithm. An explanation of how this algorithm is used to reorder inputs for BDDs is included. The concept of using symmetry to reorder inputs is also discussed.

2.1 Definition of Output Probability Algorithm

An ordering metric is a numerical assignment given to an input that indicates where it is positioned in a list of inputs. For example, a crude way of using ordering metrics might be to assign a "1" to the input that occurs first in the input ordering, a "2" to the second input in the ordering, and so on. When assigning ordering metrics to inputs, it is important that the metrics indicate some relationship between the inputs and the outputs. There are many methods that can be used to establish the relation between an input and an output. One methods that works well involves the use of output probabilities. The term *output probability* is used to denote the chances (likelihood) that an output will yield a logical one. This likelihood is based on the probability that an input will have a value of 0 or 1. In the case where all inputs are equally likely to have a value of 0 or 1, the output probability can be calculated as the percentage of 1's in the truth table representation. In this work, output probabilities are used as the basis for assigning ordering metrics to inputs. The output probability of a function will be denoted as $p\{f\}$.

In [13], output probabilities were calculated using algebraic and circuit topologies. Other methods have also been developed, such as those described in [8] [20] [21] [22] [23] [24]

[25]. Of these methods, [8] offers an efficient means of calculating output probability values. This method requires the use of a BDD as an input representation for the probability algorithm. The algorithm performs a depth-first traversal of the BDD, starting from the terminal nodes and working back towards the root node. As each node is visited during the traversal, it's *path probability* is computed. The path probability of a node is the probability that a path from that node will lead to the 1 terminal. Also, the path probability of a node can be defined only in terms of its successor nodes. Because of this, the two terminal nodes have special path probability definitions since they have no successor nodes. The path probability of a 1 terminal is 1.0, and the path probability of a 0 terminal is 0.0. Based on these two values, the path probabilities of all the other nodes in a BDD can be computed. In this computation, the following notation is used:

- PT_n The path probability of a node, n.
- PT_1 The path probability of the node referenced by the 1-edge.
- PT_0 The path probability of the node referenced by the 0-edge.
- p_n The probability of the variable associated with the node, n.

Using this notation, the path probability of a node can be defined as:

$$PT = (p_n * PT_1) + ((1 - p_n) * PT_0) \quad (2\text{-}1)$$

This equation was used to calculated the path probabilities of the nodes from Figure 2-1.

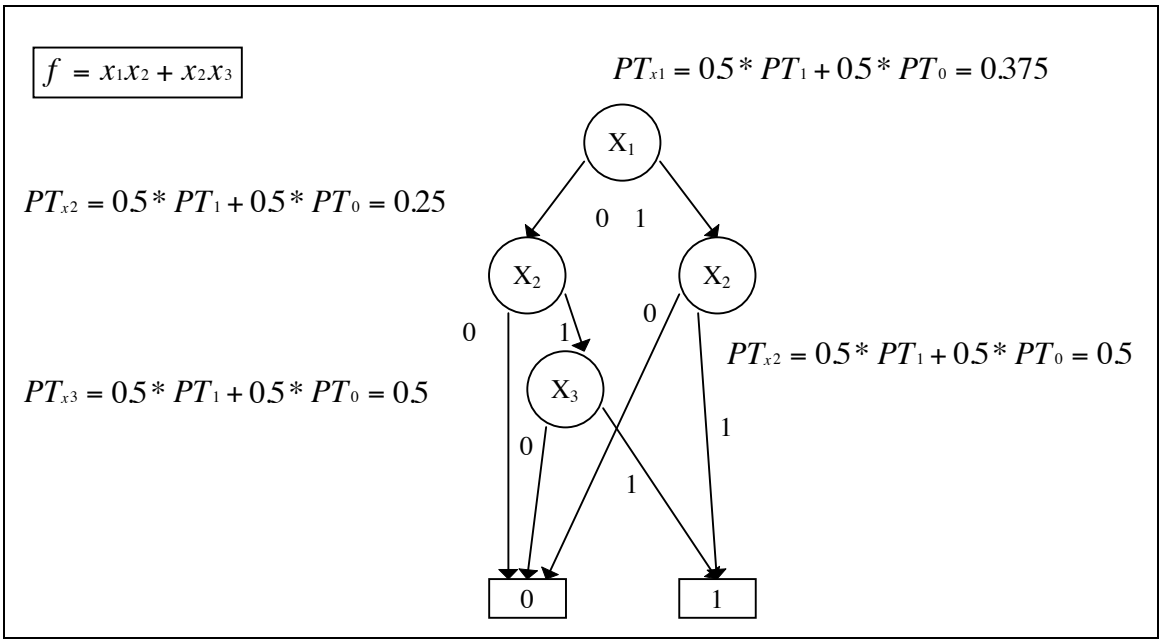

Figure 2-1: Example of a Depth-First Calculation

Notice that in this example that all of the p_n values are 0.5. Unless stated otherwise, it is assumed that an input for any given function is equally likely to take on a value of 0 or 1. Also note that the path probability of the root node is 0.375, which is also the output probability for this particular function. In other words, there is a 37.5% chance that this particular function will evaluate to a logical one. Calculating output probabilities in this manner is very efficient in terms of speed [8]. During the path probability calculation process, nodes are visited only once, which allows for a linear complexity of O(N) where N is the number of nodes in the BDD.

This method is advantageous in other aspects too. First, the probability algorithm can be used with SBDD data structures. Since a node is visited only once during a traversal, it

follows that shared sub-graphs of SBDDs are visited only once during a traversal. This maintains the complexity of the algorithm as $O(N)$. Figure 2-2 illustrates how an SBDD is used with the algorithm.

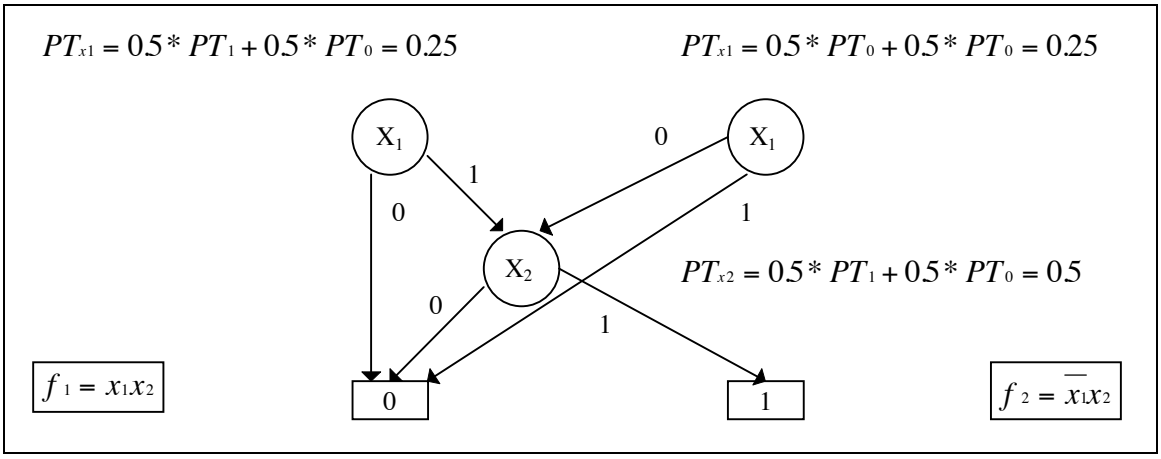

Figure2-2: Example Calculation with SBDD

A second advantage of the algorithm is that it can be modified to handle negative-edge attributes. If the attribute rules set forth in section 1.2.3 are observed, then only 0-edges can have a negative attribute. Equation 2-1 can then be modified to form Equation 2-2.

$$PT_n = (p_n * PT_1) + ((1-p_n) * (1 - PT_0)) \quad (2\text{-}2)$$

The algorithm and Equation 2-1 can be modified to handle other attributes, as well, such as positive-edge attributes.

2.2 Characteristics of Good Ordered BDDs

Creating a variable ordering for any given BDD can have a great impact on the size of that BDD. For a BDD that has n primary inputs, the size can vary from a best case of n nodes to a worst case of $2^n/n$ nodes [6]. The development of a method that determines the best order would be ideal. However, it has been proven that determining the best order for any given BDD is a NP-Complete problem [7]. Hence, the best order for a BDD probably cannot be produced in polynomial time. So far, the most efficient algorithm for determining the optimal order for any given BDD has a complexity of $O(n3^n)$ [7]. Thus, it is more practical to use heuristics to determine "good" input orderings. Recall that "good" orderings are defined as orderings that produce either the optimal size or a size that is close to optimal in a short amount of time.

2.3 Relation Between Variable Order and Output Probability

In this work, probability based metrics are used to produce good orderings. There are, however, a number of different functions that can be used in probability calculations. For example, the work presented in [8] uses the probability of the consensus function to assign ordering metrics to the primary inputs. The heuristic developed in [8] also uses the concept of weighted sum metrics. Specifically, the probability of the consensus for a particular input was taken with each output and summed together. The sum of these probability values was assigned as the ordering metric for that input.

In the work presented here, it is suggested that probability metrics need to be calculated about a single output. The idea is that one output from a set of outputs will produce better probability-based metrics than using all of the outputs. In addition to the consensus function, the Boolean XOR and Boolean AND functions were used to investigate this notion of using only a single output for creating probability-based metrics.

2.4 Relation Between Output Probability and Symmetry

Figure 2-3 illustrates a series of probability calculations plotted on a chart. These calculations were performed for the benchmark circuit c432 and were taken about the output labeled as 370gat. The y axis indicates the calculated probability values and the x axis indicates the primary inputs.

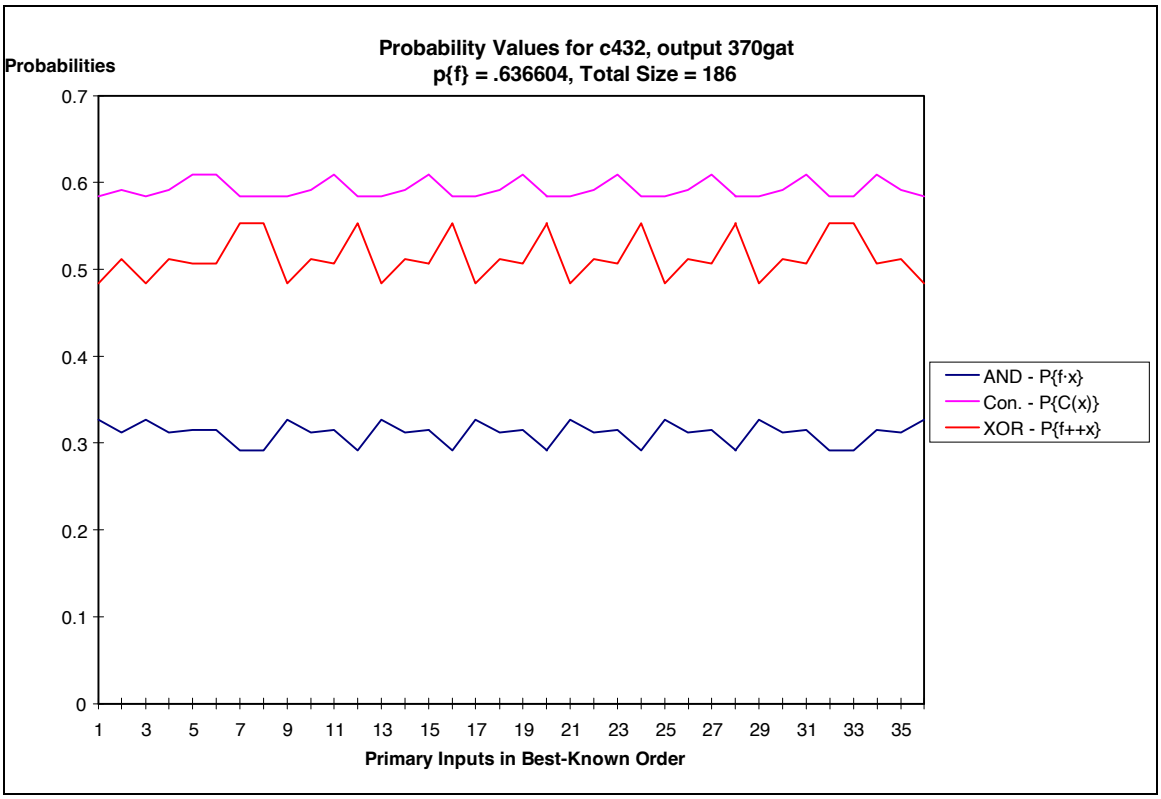

Figure2-3: Probability Values for Circuit c432

Note that the probability values are for the AND, XOR, and the consensus functions. Also note that the inputs have been arranged in their best known order; these orders were obtained from [14]. This was done so that patterns in the probability graphs could be detected. Detecting and analyzing such patterns is useful for the purpose of establishing a basis for a reordering heuristic. In this instance, an evident pattern in the probability plots exists for the XOR function. Notice that the XOR plots exhibits a periodic curve throughout the middle portion of the graph. Such periodic curves were evident on other outputs for the circuit c432.

Witnessing the periodicity in the curves for the XOR plots led to the idea of reordering variables based on their symmetry. Symmetric variables have been used before as a basis for reordering techniques [10] [27] [28] [29] [30]. Symmetric variables are interchangeable. That is, they can be switched with each other without changing the properties of the function that they are used in [26]. As a simple example, consider the inputs to any Boolean logic gate. The names (labels) of these inputs can be interchanged in any possible combination, but the result generated by the gate is not changed. However, the inputs for an adder circuit cannot necessarily be switched. If one of the operand signals was exchanged with the carry signal for a particular bit, then the resultant sum and carry signal for that bit might be altered, causing the adder to be faulty. Theorem 1 is provided to help establish a mathematical foundation between symmetric variables and output probabilities.

Theorem 1 $p\{f \square x_i\} = p\{f \square x_j\}$ *if* $x_i, x_j \square S$ where S is the set of all independent variables that support f and $x_i \square x_j$ (that is, x_i and x_j are symmetric in f).

Proof: From the Shannon Expansion Theorem:

$$f = \overline{x_i} \square f_{\overline{x_i}} + x_i \square f_{x_i}$$

Assume that x_i and x_j are statistically independent and that they are temporally and spatially uncorrelated. Also assume that they are equally likely to be a "0" or a "1" ($p\{x_i\} = p\{x_j\} = 0.5$). The properties of probability theory can then be used to make the following evaluations:

$$p\{f_{x_i}\} = p\{f \mid x_i\} = \frac{p\{f \cap x_i\}}{p\{x_i\}} = 2p\{f \cap x_i\}$$

$$p\{f \oplus x_i\} = p\{f\} + p\{x_i\} - 2p\{f \cap x_i\} = p\{f\} + p\{x_i\} - p\{f_{x_i}\}$$

Therefore:

$$p\{f\} + p\{x_i\} - p\{f_{x_i}\} = p\{f\} + p\{x_j\} - p\{f_{x_j}\}$$

$$p\{f \oplus x_i\} = p\{f \oplus x_j\}$$

•

Interestingly, the XOR plots suggest that symmetric variables, as defined by Theorem 1, should be placed as far apart as possible to produce the best order. This is contradictory to claims made by Minato that related variables should occur close to each other in the ordering. In fact, the evidence of periodicity in the XOR plots led to the development of the reordering heuristic presented here, based on enforcing the periodicity of symmetric inputs.

2.5 Algorithm for Enforcing Periodicity

With the notion that good orderings exhibit periodicity in their probability values, a method that enforces periodicity was developed, which makes use of probability

histogram charts. A probability histogram is simply a histogram that tallies the number of inputs that have distinct probability values Once a probability metric has been calculated for an input, that input can be stored in a probability histogram. An example of such a histogram is presented in Figure 2-4.

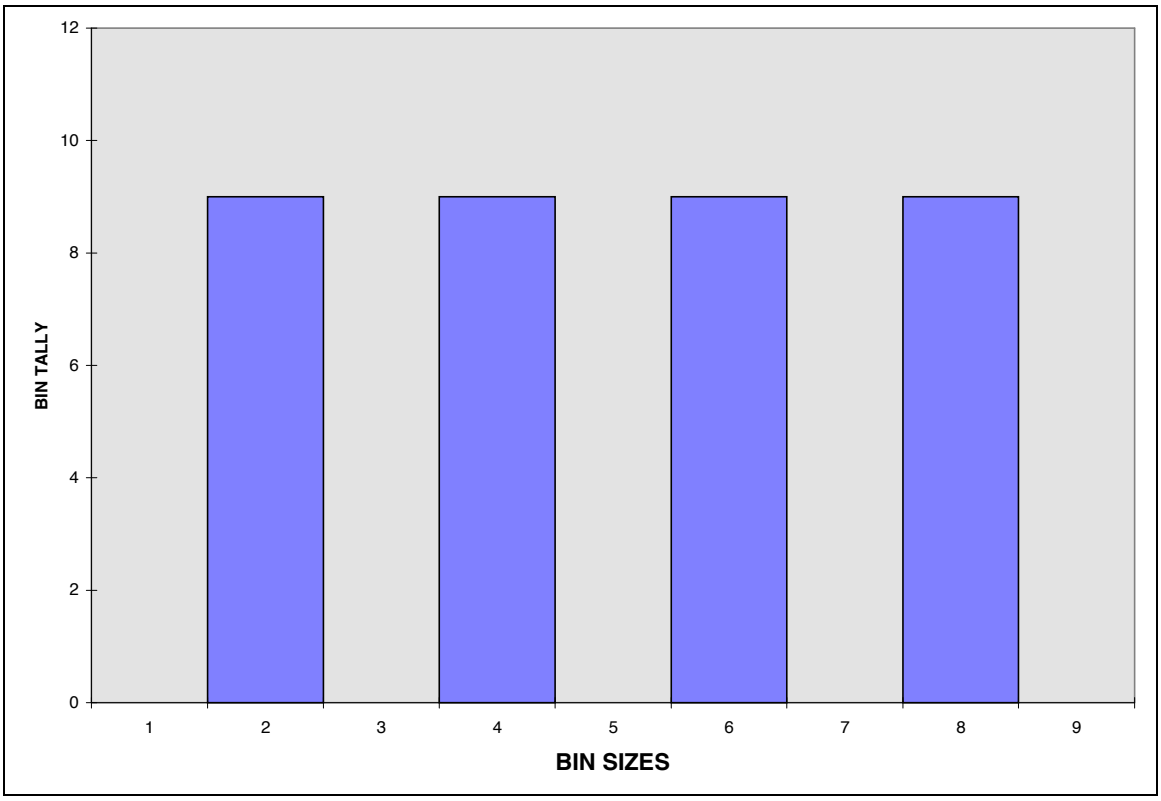

Figure2-4: Probability Histogram for Output 370gat for Circuit c432

This probability histogram is used to represent the XOR probability values that were calculated from output 370gat for the circuit c432. Note that the histogram has four bins and each bin represents a different probability value; in this instance, the probability values are: 0.483823, 0.50705, 0.512001, and 0.552911. Also note that the size for each

bin is nine. That is, each bin stores nine inputs whose probability metric values are equivalent to the label of their respective bins.

To construct such a histogram, all of the $p\{f \square x_i\}$ values for a given output must be calculated. The probability plots for the AND function also tend to exhibit some periodicity, but the range of the probability values is not as great. To create a more flexible probability histogram, a greater range of values is needed. Thus, the use of the XOR function in calculating probability values is favored over the AND function. Once the probability values for the inputs have been evaluated, they are used to determine the number of bins and the width of each bin.

Once all of the inputs have been stored, a periodic input list used to reorder the BDD can be created via a probability histogram traversal. To perform a histogram traversal, a bin is selected as the starting point. An input is removed from the starting bin and stored in an order list. The starting bin is usually either the bin that stores the inputs with the smallest probability values or the bin that stores the inputs with the largest probability values. Since probability histograms tend to have more inputs stored in the middle portion of the histogram, using the largest or the smallest sized bin as the starting point has the effect of "weeding out" inputs with uncommon probability values early in the ordering. Thus, histogram driven input orders tend to be more periodic towards the end of the list. To continue the traversal, the next adjacent bin is visited and an input is removed and stored in the order list. This process of visiting the next adjacent bin and removing and storing inputs is continued until all the bins have been visited. Then, the

process starts again at the starting bin and continues in a cyclic fashion until all of the inputs have been removed from all the bins. If a certain bin is emptied of all its inputs, yet other bins still contain inputs, it is merely skipped over. Once the traversal is finished, an input list that reflects periodicity has been created.

Consider the histogram in Figure 2-4. It can be used to create the graph represented in Figure 2-5 via a histogram traversal. Take note of the periodicity of the values in the probability plots.

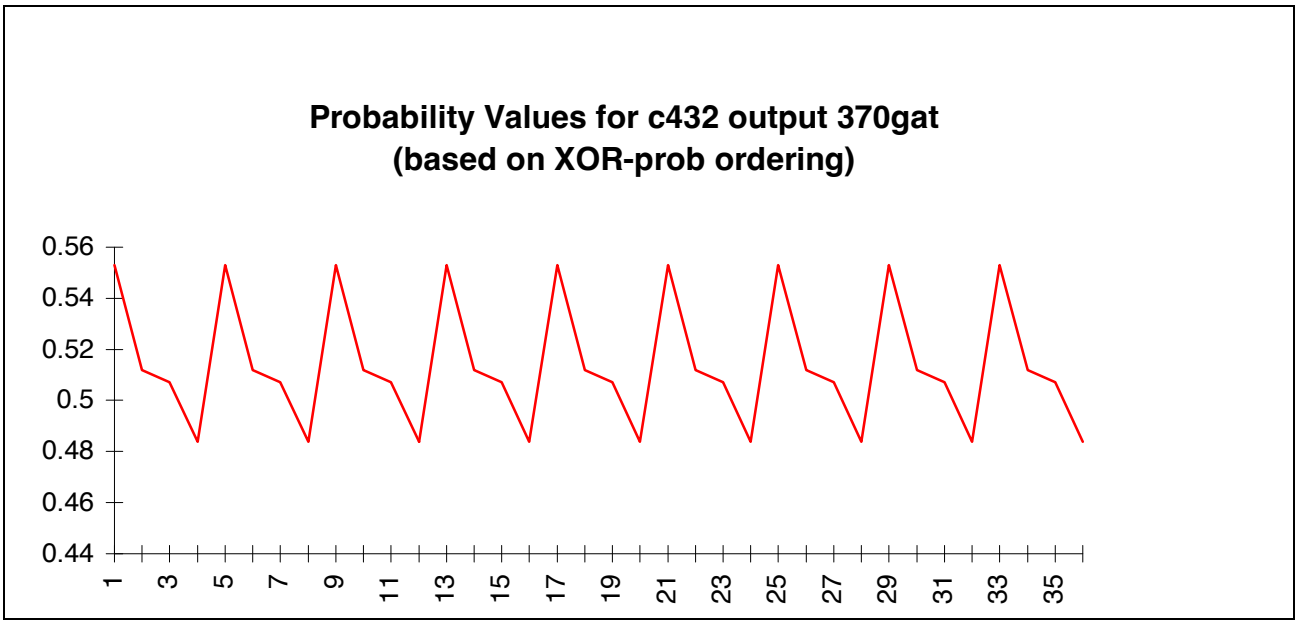

Figure2-5: Example of Histogram Driven Probability Plots

A couple of items concerning enforcing periodicity should be illustrated. First, a periodic ordering generated via a histogram traversal can be generated in four different ways. The first two involve the method by which the histogram is traversed. As in the previous

example, a histogram can be traversed in ascending order (moving from the smallest bin to the largest). Or, a histogram can be traversed in descending order (moving from largest to smallest). The list of inputs generated by both of these methods can then be reversed to create two more ways in which a histogram can be used to generate an input order. Though each of these methods enforce periodicity, they rarely return orderings that are the same and, therefore, usually do not create BDDs of the same size. The second thing to note is that orderings are periodic only if their probability values are periodic. Figure 2-4 demonstrates a case where the ordering is totally periodic since there are only four unique probability values distributed evenly over thirty-six inputs. However, if there was a wide range of probability values, then there would be little or no periodicity to enforce.

Chapter 3: Reordering Technique Defined

In this chapter, a variable reordering heuristic that will reduce the size and complexity of this data structure is presented and explained in full detail. This heuristic is based on the arrangement of symmetric input variables.

3.1 Review of Symmetry and its Relation to Reordering

Chapter 2 explains how symmetric variables can be arranged to form "good" orderings, which can be used to produce more compact BDDs. This arrangement requires that symmetric variables are placed periodically within the ordering. That is, symmetric variables are placed as far apart from each other as possible. This is unusual since other methods focus on placing related variables close to each other [6] [8] [10] [27] [28] [29] [30]. However, when placing inputs in their best-known order, empirical evidence suggests that "good" orderings exhibit a sense of periodicity. From this observation, the heuristic presented here was developed.

3.2 The Reordering Heuristic Defined

The are several steps in the technique that are used for reordering a set of inputs. In Chapter 2 it was discussed that output probabilities are useful in assigning ordering metrics to inputs. Other techniques have been developed that use all of the outputs of a SBDD for probability calculations [8]. In this work, however, only a single SBDD output

is used. This has a couple of benefits. First, using a single output to form probability values reduces the number or required calculations and, therefore, increases the speed of the technique. Secondly, if a SBDD has *m* outputs, it is likely that at least one output will provide better probability values than the others. Thus, selecting an output to use for the probability calculations is a critical issue.

After an output is selected, it is used to calculate the probability metrics for all of the inputs. The function used to compute these values is based on the Boolean XOR function. It can be expressed as $p\{f \oplus x_i\}$. The value returned by this calculation is the metric that is assigned to the respective input. For example, if $p\{f \oplus x_2\} = 0.5$ for the input variable x_2, then the ordering metric for x_2 is 0.5.

Once all of the $p\{f \oplus x_i\}$ values have been calculated, a probability histogram is constructed based on the $p\{f \oplus x_i\}$ values. The probability histogram contains a certain number of bins. Each bin has the properties of size and width; size can also be thought of as the label, or name of the bin. For example, if a certain bin is labeled as 0.5 and also has a width of 0.5, it will store any input whose probability metric has a value between 0.5 and 1.0. Since probability histograms are used to generate a list of input orders, it is imperative that the histogram is constructed in a manner that produces the best, possible ordering. Ascertaining good values for the bin sizes and widths is somewhat difficult. Varying the total number of allowable bins and the range of inputs they can store has varying results. In some cases, better orderings are produced when only a few bins with wide ranges are created. In other cases, better orderings can be achieved if numerous bins

with narrow ranges are used. Since our focus is on periodic orderings, the probability histograms used in this heuristic have been made so that the bin widths are as narrow as possible. That means that a bin is created for each distinct probability value and that only inputs with equivalent probabilities are stored in the same bin. Due to memory constraints, a threshold value of 10,000 has been defined. Thus, should the number of bins exceed 10,000, then there might be instances where inputs with varying probability values are stored in the same bin. A probability histogram can be created in the following steps:

1. Find the two closest valued $p\{f \square x_i\}$ values that are not equal and compute their difference, s.

2. Find the difference, b, that corresponds to subtracting the overall smallest $p\{f \square x_i\}$ value from the largest. Hence, b is the range of probability values.

3. Set the number of bins (BINNUM) = $(1 / s)$.

4. If BINNUM exceeds a threshold value, set it to the threshold.

5. Set the size of each bin (BINSIZE) = b / BINNUM.

For example, suppose that a function has five inputs named a, b, c, d, and e. The five probability values are 0.1, 0.2, 0.3, 0.1, and 0.2, respectively. Since there are only three distinct values (0.1, 0.2, and 0.3), only three non-zero bins are created. There are two three bins that store more than one input. The first bin has a size (label) of 0.1. It's width is narrowed so that only inputs with a metric equal to 0.1 are stored within it. A bin for 0.2 and a bin for 0.3 are also created.

When the bin sizes and widths have been determined, the probability histogram is filled. This is a simple process in which inputs are placed in the histogram according to their metric value. Continuing the example in the paragraph above, inputs *a* and *d* have a metric value of 0.1 and are stored in the bin whose size is 0.1 while inputs *b* and *e* are stored in the bin with size 0.2 and input *c* is stored in the bin with size 0.3.

In order to traverse the histogram and obtain an ordered list of inputs, a starting point must be selected. Theoretically, any bin may be chosen as the starting point, however, it is more practical to select either the leftmost bin or the rightmost bin as the starting point. Note that the leftmost bin stores inputs with the smallest probability value and the rightmost bin stores inputs with the largest probability value. In addition to selecting the starting point, the method by which an input is removed from the bin must also be chosen. We have experimented with three different ways in which to choose an input from a bin. The first is a random selection, the second and third are queuing methods (first in first out, or FIFO, and last in last out, or LIFO). Each of these methods have been tested and evaluated against each other. Both the LIFO and FIFO method yield better results. This is probably due to the preservation of the compactness provided by the initial BDD ordering. However, there is no clear indication whether a LIFO or a FIFO method is better at selecting an input from a bin. For certain cases, the LIFO approach works better and vice versa. In the work presented here, the LIFO method for selecting inputs was used.

With a starting point and a method for selecting inputs, the histogram can then be traversed in a circular fashion. There are two ways in which the histogram can be traversed so that periodicity is enforced. The first method is called the ascending approach. In an ascending approach, bins are visited in ascending order with accordance to their size. For example, the bin 0.1 would be visited. Next, bin 0.2 would be visited, then bin 0.3; this can also be though of as a left-to-right traversal. Bear in mind that as each bin is visited, an input is taken from it and stored in an input list. The second traversal type is the descending approach, or the right-to-left traversal. Such a traversal can be illustrated as starting at the 0.3 bin, then moving to the 0.2 bin, and then to the 0.1 bin. Regardless of which traversal type is chosen, the histogram must be continually traversed until all of the inputs have been emptied from all of the bins and stored in an order list. If the inputs are not distributed evenly among the bins, then certain bins will be emptied before others. In such cases, bins with size zero are simply overlooked during the traversal. Again, consider the case where there are three bins: 0.1, 0.2, and 0.3. Suppose during the first pass that the 0.1 bin is emptied, but the 0.2 bin and the 0.3 bin each have three inputs left. The traversal would continue for three more passes, removing an input from the 0.2 bin and then the 0.3 bin (or the 0.3 bin and then the 0.2 bin, depending on the traversal type) while overlooking the 0.1 bin.

After the traversal of the probability histogram is completed, a new input ordering has been created which reflects the amount of periodicity in the input metrics. For example, suppose that a probability histogram had only 2 bins, one with a size of 0.4 and one with a size of 0.6. Also suppose that each bin contained four inputs. The resultant ordering

then would have eight inputs whose metric values would alternate between 0.4 and 0.6. However, it is not usually the case that inputs are evenly distributed along the probability histogram. Using the input order generated by the traversal of the probability histogram, the SBDD can be reordered so that its input sequence is identical to the histogram driven order.

The heuristic used to enforce periodicity can summed up in the following steps:

1. Select an output about which to compute n values of $p\{f \mid x_i\}$.
2. Compute the probability values.
3. Determine histogram bin sizes and widths.
4. Construct a histogram using the $p\{f \mid x_i\}$ values.
5. Starting at the bin with the lowest (or highest) probability value, remove a value and store it in an order list.
6. Move to the next adjacent bin (in a cyclic fashion), remove a value and store it in the order list.
7. Repeat step 6 until all values have been removed from the histogram and stored in the order list.
8. Use the order list to reorder the initial SBDD.

3.3 Analysis of the Heuristic

This section presents some experimental results that were gathered by using the reordering heuristic described in the previous section. These results were generated on a

100 MHz Sun SPARCstation 20 with 240 MB of RAM, running under Solaris 2.5. The heuristic was implemented with C code using the GNU gcc compiler. A BDD package developed by David Long was used with the code. The code uses the *ISCAS*85 benchmark circuits for the input functions.

Table 1 displays some of the experimental results. The column labeled *Original Size* indicates the size of the SBDD initially. The initial variable ordering were the orderings generated by a depth-first traversal of the benchmark circuit netlists. The column labeled *Sift Size* indicates the size of the SBDD representation after it has been condensed via Rudell's sifting method. This data is included as a metric to compare our results with. The column labeled *Descending* indicates the size returned by using our reordering technique if the probability histogram is traversed in a descending (right-to-left) fashion. Thus, the column labeled *Ascending* indicates the returned size if an ascending traversal (left-to-right) is used. Run times have also been included.

Circuit Name	Org. Size	Sift Size	Sift Time	Descending	Time	Ascending	Time
c432	31172	20891	28	2317	6	1841	6
c499	53866	38196	84	1328755	463	1328755	461
c880	10348	5089	9	549791	447	166573	110
c1908	13934	8175	11	19039	12	19077	13

Table 1

The results presented in Table 1 are mixed. A SBDD that has only 1841 nodes can be constructed for c432. This figure is approximately an order of magnitude smaller than the size returned by Rudell's sifting technique. Yet when examining results for other circuits, such as c499, the result is not very good. In fact, the heuristic causes the SBDD

representation for c499 to grow rather large, roughly two orders of magnitude greater, a very undesirable effect.

There are, however, still two more methods by which a periodic ordering can be generated by traversing the probability histogram. There is the option to *reverse* the order that is produced by either traversal method. Hence, there are four types of periodic orderings that can be created by the heuristic: descending, descending-reversed, ascending, and ascending-reversed. The concept of reversing the histogram driven orders came by comparing the orders against the best known orders. In many cases, the histogram driven orders bore a striking resemblance to what the best-know orders would be *if they were in the reverse order*! Table 2, therefore, demonstrates the effects of taking the histogram driven input order and reversing it.

Circuit Name	Org. Size	Sift Size	Sift Time	Descending	Time	Des-Rev.	Time	Ascending	Time	Asc-Rev.	Time
c432	31172	20891	28	2317	6	1841	5	1841	6	2733	6
c499	53866	38196	84	1328755	463	54202	23	1328755	461	54201	25
c880	10348	5089	9	549791	447	208686	121	166573	110	208898	217
c1908	13934	8175	11	19039	12	19159	17	19077	13	11850	18

Table 2: With Reversed Orders

In almost every instance, reversing the histogram driven order produces better results than not reversing the order. Still, the results are not very promising. Bear in mind that while reversing the histogram driven order can produce orders that are *similar* to the best known orders, they rarely produce orders that are *identical* to the best known orders. Thus, "good" orders are not guaranteed by reversing histogram driven orders.

The technique presented in [8], which also uses probability metrics to reorder inputs, has been shown to act well as a pre-sifting filter. Since other probability based reordering heuristics work better by incorporating sifting, it was theorized that the heuristic presented here might also work better as a pre-sifting filter rather than a stand-alone technique. Table 3 presents the results of using our heuristic as a pre-sifting filter. The style of sifting used to gather the data in Table 3 is called *iterative sifting*, or ITR sifting. The iterative sifting approach differs from standard sifting in that a recursive series of sifts is applied to an input order. For example, standard sifting searches a n^2 set of inputs and returns the ordering that produces the best size. Iterative sifting performs the same process repeatedly until an order that produces a SBDD of identical size as the previous iteration is returned. This is the signal that any further iteration of the sifting process will have no effect. In Table 3, the columns labeled Descending, Des-Rev., Ascending, and Asc-Rev. each incorporate the iterative sifting process to produce the indicated sizes.

Circuit Name	Org. Size	Sift Size	Sift Time	Descending	Time	Des-Rev.	Time	Ascending	Time	Asc-Rev.	Time
c432	31172	20891	28	1065	8	1119	9	1069	8	1109	8
c499	53866	38196	84	37421	2254	36234	281	37421	2287	36241	249
c880	10348	5089	9	8566	879	14558	319	4505	494	7642	395
c1908	13934	8175	11	11850	61	10535	49	11850	62	8235	54

Table 3: With ITR Sifting

Table 3 indicates that using the heuristic as a pre-sifting filter works much better than using the heuristic by itself. In many cases, the heuristic with iterative sifting works better than Rudell's standard sifting technique. The use of iterative sifting, however, has a couple of problems. First, there are still instances in which not very good orderings are produced. Consider the results for the circuit c1908 in Table 3. In three of the instances,

the sizes are not much better than the original size of the circuit. In the fourth instance, which was produced by the ascending-reversed method, the results are comparable, but not better than sifting. In order for a reordering technique to be useful, it needs to operate well on a large percentage of circuits and functions. Another problem with using iterative sifting is that it disrupts the periodicity of the ordering. The main focus of the heuristic presented here is to enforce the periodicity of an order and demonstrate that "good" orders exhibit periodicity. Orders returned by iterative sifting are not necessarily periodic. However, the results in Table 2 indicate that, overall, histogram driven orders are not "good" orders. On the other hand, the $p\{f \mid x_i\}$ data plots suggest that "good" orders do indeed exhibit periodicity. To reconcile these two issues, a new modification to the sifting technique was developed. It is described in the following section.

3.4 Bin-Sifting

Using a probability histogram to generate a periodic ordering of inputs does not necessarily produce a "good" input ordering. In fact, there are more cases than not in which a "good" ordering is not produced. The reason for this involves the manner in which inputs are removed from the bin in the probability histogram. Three methods for removing inputs from the histogram have been used, the FIFO approach, the LIFO approach, and the random approach. Empirical evidence suggests that of the three methods, the LIFO approach generally returns the best results. This technique assumes that the best way to enforce periodicity is to read each histogram bin like a stack. Though this works well for some instances, it does not guarantee that the best , periodic ordering

will be generated. What is needed is a technique that can alter the histogram generated input order without affecting the periodicity of the input order. Such a technique can be achieved via *bin-sifting*.

Bin-sifting is a modification of Rudell's sifting technique. Like standard sifting, bin-sifting searches a sub-set of input variables and returns the ordering that would produce the smallest size. Bin-sifting, however, differs from standard sifting by selecting a smaller subset of variables to sift. That is, bin-sifting is applied only to variables that have the same $p\{f \square x_i\}$ values. Hence, only inputs that are stored within the same histogram bin are sifted with each other. As an example, consider the probability histogram illustrated Figure 3-1.

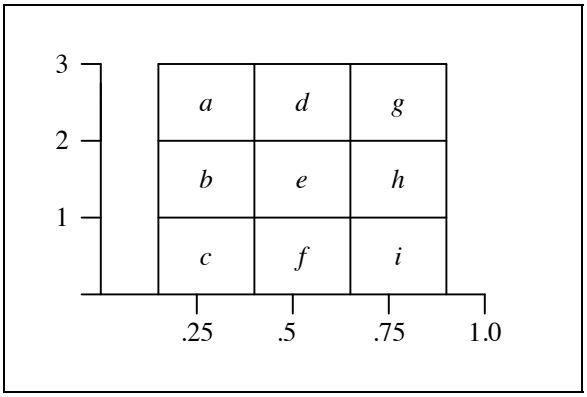

Figure 3-1: Example Probability Histogram

In this example, there are three bins. Each bin stores three inputs. By performing an ascending traversal of the histogram, an input order of *a d g b e h c f i* is generated. Suppose that the best order for these nine variables was *c d g a e h b f i*. Bin-sifting can be used to obtain this ordering by sifting only the variables from the first bin, which are *a b* and *c*. Thus, after the bin-sifting algorithm is completed, the new order becomes *c d g a*

e h b f i. Note that the only variables whose positions were altered in the order had the same $p\{f \square x_i\}$ values.

Applying bin-sifting to a histogram driven input order never degrades the current SBDD size and almost always improves the order since it is a "greedy" technique. To ensure that the best possible order is created, the bin-sifting technique must be applied to every bin. Thus, if there are five bins in a probability histogram, then the bin-sifting technique should be applied five times, one for each group of inputs stored within a particular bin. Typically, bin-sifting performs better than standard sifting. Thus, should one decide to reorder a BDD that exhibits little or no periodicity, one might choose to apply bin-sifting rather than standard sifting. Experimental results of the bin-sifting technique are presented in section 3.6, which includes other results for comparison and analysis.

The steps of the heuristic to enforce periodicity can now be fully listed:

1. Select an output about which to compute *n* values of $p\{f \square x_i\}$.
2. Compute the probability values
3. Determine histogram bin sizes and widths
4. Construct a histogram using the $p\{f \square x_i\}$ values.
5. Starting at the bin with the lowest (or highest) probability value, remove a input and store it in the order list.
6. Move to the next adjacent bin (in a cyclic fashion), remove a input and store it in the order list.

7. Repeat step 6 until all inputs have been removed from the histogram and stored in the order list.

8. Use the order list to reorder the initial SBDD.

9. Apply bin-sifting, if needed.

3.5 Example of the Method

To provide a better understanding of how the heuristic works, this section provides a simple, step-by-step example. To illustrate how the technique works, the "toy benchmark" circuit c17 has been selected. Since the purpose of this example is to demonstrate the functionality of the heuristic, a ROBDD data structure rather than an SBDD with negative-edge attributes is used for the sake of simplicity and clarity.

The first step in the method is to select an output to be used in the probability calculations. The circuit c17 has two outputs: 22gat and 23gat. In this example, 23gat is chosen. Figure 3-2 depicts how the ROBDD representation for the function/output 23gat appears in its original order. For circuit c17.isc there are five inputs, 1gat, 2gat, 3gat, 6gat, and 7gat. Notice that the input 1gat does not appear in the ROBDD diagram in Figure 3-2. That is because 1gat does not appear in the support of the output 23gat. For the ROBDD in Figure 3-2, the original variable order is 2gat, 3gat, 6gat, 7gat. Also take note that the size of this ROBDD is 5 non-terminal nodes.

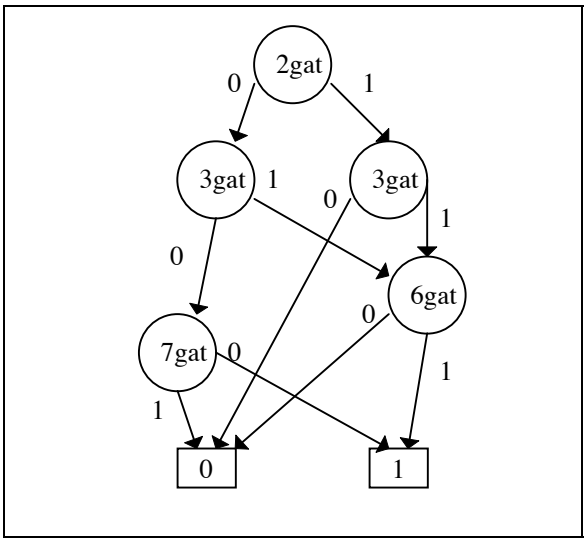

Figure 3-2: ROBDD in Initial Variable Order

With 23gat as the chosen output, the $p\{f \square x_i\}$ probability values can be calculated. This step utilizes the depth-first algorithm described in Chapter 2. Recall that this algorithm requires a SBDD as an input, rather than an ROBDD. For n inputs, a SBDD with N nodes must be traversed n times to calculate all of the probability values. In this instance, the value of n is five. Since the value of N can range from n to 2^n, the worst case complexity of calculating these probability values is $O(n2^n)$. Reordering, however, is usually not performed on BDDs that have an exponential number of vertices since such representations are difficult to construct. In this case, the SBDD for c17 is small and N is equal to 5. Hence, reordering this circuit is neither difficult nor time consuming and the $p\{f \square x_i\}$ values can easily be formulated. These values are:

- $p\{f \square x_2\} = 0.6875$
- $p\{f \square x_3\} = 0.3125$
- $p\{f \square x_6\} = 0.6875$
- $p\{f \square x_7\} = 0.3125$

The third step of the heuristic calls for a probability histogram to be created. The probability values are used to determined the number of bins and the widths of the bins in the histogram. To do this, the two closest $p\{f \square x_i\}$ values are taken and the smaller one is subtracted form the larger one. In this case, 0.3125 is subtracted from 0.6875. This difference, called *s*, is equal to 0.375. Next, the range in values from the smallest probability to the largest probability value is calculated. Thus 0.3125 is subtracted from 0.6875 to get 0.375, which is denoted by *b*. The next step involves calculating the number of possible bins that can appear in the histogram. This is done by dividing 1 by the value *s*. This evaluates to 2 and 2/3. Obviously, 2/3 of a bin can't be created, so the number of bins is rounded to 2. Should the number of bins exceed a threshold value, then the number of bins is set equal to the threshold; for this heuristic, the threshold has been set to 10,000 and, clearly, there is no danger of exceeding this limit. The width of the bins can be calculated by taking the value *b* and dividing it by the number of bins. Dividing 0.375 by 2 results in a bin width of 0.1875. It has been stated that the bins in the probability histogram are created so that only inputs with similar probability values are stored in the same bin. Thus, it might seem contradictory to have two bins with a width of 0.1875 each. However, since the value of the bin width is less than the value of *s*, it is impossible for inputs with varying probability metrics to be stored in the same bin.

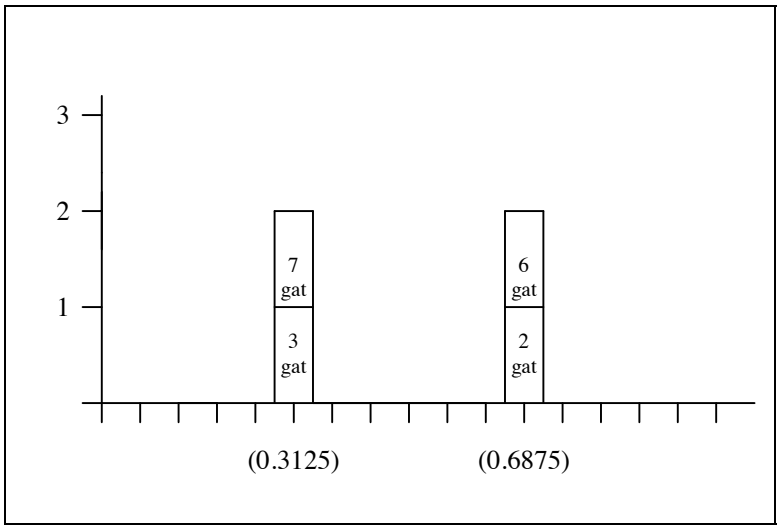

Figure 3-3: Probability Histogram for ROBDD of 23gat

Once the number of bins and their widths have been determined, the probability histogram must be filled. Figure 3-3 is an example of how this is done. Notice that there are 2 bins depicted, which was the number calculated in the third step of the algorithm. One bin is for the variables that have a metric value of 0.3125 and one for the variables that have a metric value of 0.6875. Observe that the variables are stored in the histogram in the order in which their metric values were defined. For example, the metric value for 3gat was calculated before the metric value for 7gat was calculated. Thus, 3gat is stored in the 0.3125 bin before 7gat is; this can be evidenced by the fact that 7gat is "on top" of 3gat in the probability histogram.

The fifth step of the technique involves selecting a starting position and removing an input. There are only two cases for selecting a starting position, either the bin with the smallest size or the bin with the largest size. In this example, the smallest sized bin,

0.3125, is selected and the input 7gat is selected. Once 7gat has been selected, it is stored in an order list which will be used to generate a new ROBDD representation.

Once a starting point has been selected, the traversal method is determined. By selecting the smallest sized bin as the starting point, an ascending traversal will be performed. That is, the probability histogram will be traversed from left to right in a circular fashion. Each time a bin is visited, an input is removed and stored in the input list. Thus, after removing 7gat from its bin, the bin 0.6875 is visited and 6gat is removed and stored in the list. This completes one pass of the histogram. To continue in a circular fashion, bin 0.3125 is re-visited and 3gat is removed and stored in the list. Then, bin 0.6875 is visited and 2gat is removed and stored in the list.

The next step uses the histogram driven order list to reorder the SBDD. In this example, the new order is 7gat, 6gat, 3gat, 2gat. Notice the periodicity of the metrics in this new order: 0.3125, 6875, 0.3125, 0.6875. Figure 3-4 illustrates the effect of rearranging the SBDD to reflect the new, periodic order.

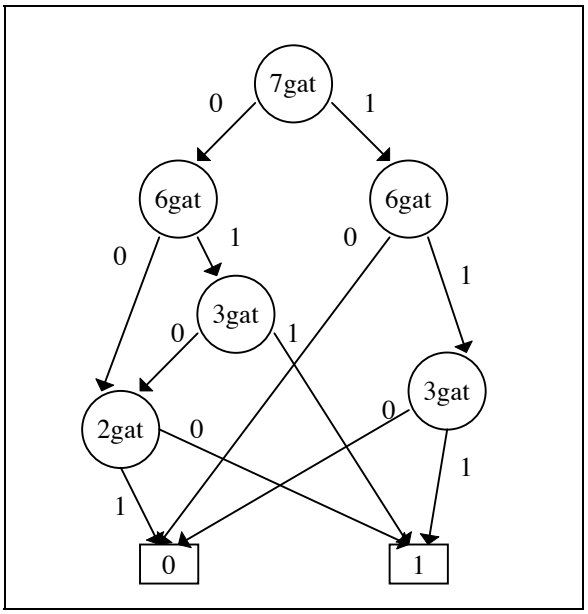

Figure3-4: Example of the Histogram Driven ROBDD

Notice that the ROBDD representation in Figure in 3-4 has six non-terminal vertices while the ROBDD in Figure 3-2 has five non-terminal vertices. Though the periodicity of the new ordering has been enforced by traversing the probability histogram, it is a worse ordering than the initial one. To improve the ordering while maintaining its periodicity, bin-sifting is applied.

The bin-sifting process visits each bin in the probability histogram and sifts the variables within a common bin. In the first bin, there are only two variables, 7gat and 3gat. The bin-sifting algorithm takes these two variables and examines a n^2 set of orderings. In this instance, there are only two distinct ways to arrange 7gat and 3gat; there is their original positioning, or they can be swapped so that 3gat appears before 7gat. Thus, the input order can remain 7gat, 6gat, 3gat, 2gat, or it can be altered so that it becomes 3gat, 6gat, 7gat, 2gat. Bin-sifting returns the order that produces the smaller ROBDD. In this case,

the order 3gat, 6gat, 7gat, 2gat is the better ordering. The ROBDD generated by this ordering is presented in Figure 3-5.

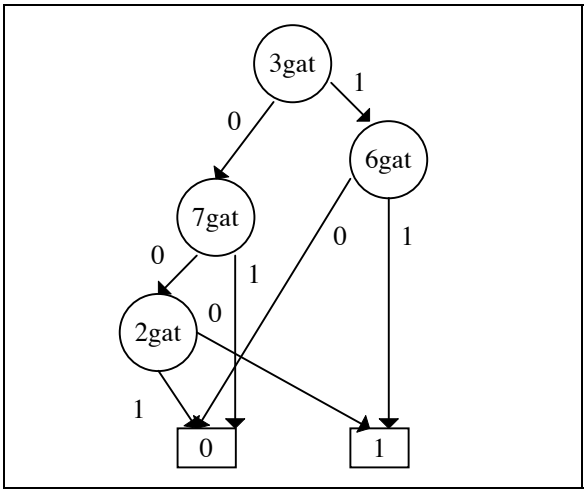

Figure 3-5: Example of the ROBDD after Bin-sifting

The ROBDD in Figure 3-5 contains only 4 non-terminal nodes, which is smaller than the ROBDD size for the initial order. Since the number of inputs for the output 23gat is 4, then the ROBDD representation is Figure 3-5 is optimal and, therefore, there is no need to continue the application of bin-sifting. Thus, this example demonstrates how the histogram presented here can generate "good" orderings by enforcing periodicity. The results of this example are summarized in Table 4. As a reminder, n is used to denote the number of inputs and N is used to denote the number of nodes.

EXAMPLE - RESULTS SUMMARY

BDD SIZE

- Original Size: 5 Vertices
- Histogram Driven Size: 6 Vertices
- Bin-Sifted Size: 4 Vertices

COMPUTATION

- 4 Probability Computations: $O(n \cdot N)$
- Histogram Formation: $O(N)$
- Bin-Sifting: $O(N^2)$

Table 4: Summary of Example Results

3.6 Experimental Results

This technique has been used to generate a number of orderings for certain benchmark circuits using a 100 MHz Sun SPARCstation 20 with 240 MB of RAM running under Solaris 2.5. The code used to implement the reordering method was written in C and compiled using the GNU gcc compiler. It should also be noted that an output to be used in the probability calculations had been pre-selected prior to the simulation of the heuristic. Thus, this step of the heuristic is not reflected in the timing measurements.

Table 5 contains the experimental results of our heuristic compared against Rudell's standard sifting technique [10]. The sifting technique is useful for comparison because it can reduce the size of a SBDD in a short amount of time. The circuits used in Table 5 are

from the set of PLA benchmarks. In this table, the column labeled Circuit Name indicates the name of the benchmark circuit. The column Original Size indicates the size of the SBDD representation when the circuit is read in its original order. The next two columns display the resultant SBDD size when Rudell's sifting technique is used and the time involved in reducing the size of the SBDD. The next two columns indicate the resultant SBDD size and the time it takes to generate that size when our technique is used.

Circuit Name	Original Size	Sifting Size	Sifting Time	Prob. Meth. Size	Prob. Meth. Time
5xp1	74	51	<1	42	<1
alu4	1197	800	<1	639	1
bw	108	103	<1	99	<1
duke2	973	394	<1	339	1
misex1	41	37	<1	37	<1
misex2	136	89	<1	84	<1
misex3	1301	704	<1	683	4
sao2	155	94	<1	87	<1
vg2	1044	335	<1	93	1
misex3c	828	504	<1	472	1
clip	226	87	<1	87	<1
e64	1441	231	1	231	2
apex1	28336	1356	3	1333	58
apex2	7096	708	1	573	76
apex4	928	895	<1	889	1
apex5	2679	1130	<1	1130	4

TABLE 5: Experimental Results Using PLA Benchmarks

The results presented in Table 5 show that, in most instances, the resultant SBDD size is less than or equal to the original size; in the cases where the resultant SBDD is not smaller, it is equal to the original SBDD size. Also, the results of our heuristic return sizes less than or equal to the sizes returned by the sifting technique in comparable

amounts of time. This demonstrates that in general, our heuristic outperforms sifting and is more useful for generating smaller SBDDs in a short amount of time.

Table 6 uses the ISCAS85 benchmark circuits to present more data. The layout for Table 6 is similar to Table 5. An extra set of data, however, has been included. Note that in Table 6 there are two additional columns. These columns represent the resultant SBDD size and reduction time when the technique of bin-sifting is used alone. Including this data has two purposes. One, it demonstrates how bin-sifting compares against standard sifting. Second, it shows to what degree bin-sifting affects the overall performance of the heuristic. For this table, results for the circuits c2670 and c7552 were not included due to problems that arose in the attempt to represent them in memory; in the code used to implement the technique, a SBDD size limit was established to reduce the memory intensity of the simulations. The circuit c6288 was also excluded from the experimental data since this circuit represents a circuit that has been shown not to have a size that is less than exponential in the number of variables.

Circuit Name	Original Size	Sifting Size	Sifting Time	Bin-sift Size	Bin-sift Time	Prob. Meth. Size	Prob. Meth. Time
c432	31172	20891	28	20891	216	1119	13
c499	53866	38196	84	36234	156	32338	277
c880	10348	5089	9	4603	20	4574	288
c1355	53866	38186	88	36234	159	32338	321
c1908	13934	8175	11	7762	23	6170	48
c3540	72858	41918	73	37644	1421	51537	257
c5315	747662	4078	225	2334	506	2314	1555

Table 6: Experimental Results Using ISCAS85 Benchmarks

The results of Table 6 show that the technique generates smaller sized SBDDs than both sifting and bin-sifting for almost every benchmark. As a tradeoff, it takes more CPU time to generate these more compact sizes. However, the additional time is generally less than an order of magnitude when compared to standard sifting. Typically, bin-sifting outperforms standard sifting with little cost in CPU time; the case for c3540 is an obvious exception.

Though using the full technique results in smaller SBDD sizes, there are instances in which the resultant SBDD size is not much smaller than the size returned by just applying bin-sifting. This is usually an indication that the $p\{f \square x_i\}$ values are not very periodic. Thus, for such cases, one might prefer to apply either sifting or bin-sifting if time is a crucial factor.

The data for the reordering heuristic in Table 6 was generated by using the descending-reverse method to generate the new ordering. Generally, this method tends to provide better results than other types of traversals. However, there are some cases in which another type of histogram traversal can be used to provide even better sizes for the benchmark circuits. Since the results presented in Table 6 are not necessarily the best results that can be gathered, Table 7 is included to show the best results that can be achieved by using this heuristic. To evaluate the effectiveness of the heuristic, the best-known to date results using Simulated Annealing (SA) [11] and Evolutionary Algorithms (EA) [15] techniques have also been included.

Circuit Name	SA size	EA size	prob. meth. size
c432	1087	1064	1065
c499	25866	25866	31229
c880	4053	4053	4254
c1355	25866	25866	31229
c1908	5526	5526	6170
c2670	.	1774	.
c3540	23828	23823	38096
c5315	.	1719	2314
c7552	.	2212	.

Table 7: Comparison of Best Known Results

Both the SA and EA techniques have been used to find the best known orderings for these benchmark circuits. They are, therefore, a useful metric to gauge the effectiveness of the heuristic. As illustrated in Table 7, the heuristic developed here returns sizes that are close to the results given by SA and EA. Run times have not been included since both the SA and EA approach have considerably long run times, making a practical comparison difficult. Thus, this heuristic can return sizes that are close to the best known orders. This demonstrates that "good" orderings can be obtained by enforcing the periodicity of the probability metrics.

Chapter 4: Conclusions and Future Efforts

4.1 Conclusions

The use of BDDs in digital design CAD tools can be very beneficial. In order to be useful, however, BDDs must be as small as possible. Since speed is also a factor in using BDDs in digital design applications, reducing the size of a BDD should also be as fast as possible. To this end, a new variable reordering heuristic for SBDDs has been presented. This heuristic uses the output probability of the XOR function to assign ordering metrics to inputs. Once metrics have been assigned, the inputs are reordered to produce a SBDD of "good" size in a short amount of time. Unlike previous reordering techniques, the work presented here enforces the periodicity of the metrics. Rather than being placed close to one another, inputs that have equivalent metric values are placed as far apart as possible in the ordering. This reordering process works well as a pre-filter to a variation of the sifting technique called bin-sifting.

Chapter 2 described an algorithm that is very efficient for calculating output probabilities. As an example of its efficiency, this algorithm can be modified to handle SBDD data structures that have negative-edged attributes. It also operates with linear complexity in terms of BDD vertices. The properties of "good" ordered BDDs were explained, and the relation between probability metrics and "good" orderings was also discussed. In addition, an algorithm that enforces the periodicity of probability metrics was developed. This algorithm has been shown to be useful for providing "good" orderings.

Chapter 3 provides a detailed description of the probability-based reordering heuristic that can be used with SBDDs that have negative-edge attributes. This algorithm enforces the periodicity of the probability metrics for the inputs to create a new order for the SBDD. In some cases, the periodic orderings caused the SBDD to grow in size. In order to maintain a periodic ordering and reduce the SBDD in size, a modification to the sifting technique was developed. This modified method is called bin-sifting. Bin-sifting, like sifting, searches a sub-space of possible orderings and returns the order that creates the most compact SBDD. However, unlike sifting, bin-sifting searches a more restricted sub-space, one that is defined by the probability histogram. More specifically, bin-sifting sifts only inputs that have the same probability metric. By generating an enforced periodic order and then applying bin-sifting, SBDD of "good" size can be obtained. In many cases, the sizes returned by this heuristic are smaller than the results generated by sifting while maintaining a similar runtime. By exploiting symmetry and applying bin-sifting, this reordering technique has the advantage of creating smaller SBDDs that are independent of the original ordering. In this regard, the technique presented here is quite flexible and useful.

4.2 Future Work

As indicated in Chapter 2, only a single output is required to use in the calculation of the probability metrics. However, determining which output provides the best metrics is not an easy task and, presently, the algorithm described here is incapable of selecting such an output. Instead, the desired output has to be specified by the user.

Another problem is determining the widths and number of bins in the probability histogram. Varying the bin width affects the ordering created by the histogram traversal and, thus, affects the resultant size of the SBDD. Currently, bin widths are created so that only inputs with equivalent metrics are stored within the same bin. However, varying the bin width does not always have an adverse affect; in some cases, widening the width of the bins can help create orders that further reduces the size of the SBDD. Ideally, the heuristic should be modified so that it selects bin widths that produce the best order via a histogram traversal. Concerning the number of bins, a threshold value has been established to ensure that number of bins created is reasonable. In most cases, the number of bins does not exceed 10,000. For this reason, 10,000 was used a threshold value during the construction of the histogram. However, a value other than 10,000 might be a more appropriate choice for the threshold value.

A third problem relates to how the input orders are created via a probability histogram traversal. An input order can be created by an ascending traversal or a descending traversal. Input orders can also be created by reversing the order returned by the histogram traversal, hence the ascending-reversed orderings and the descending-reversed orderings. In most cases, a descending-reversed order yields the best results. Unfortunately, there are instances in which another type of order would be optimal. So far, a way to select the best traversal method has not been determined.

Lastly, there is the instances in which this heuristic is applied to functions that exhibit little or no periodicity. In such cases, the runtimes for applying the entire heuristic are

lengthy and might not be desirable, even though a smaller sized SBDD is sometimes obtained. In such instances, it might be preferable to apply only bin-sifting or even plain sifting. However, this heuristic has not been designed to detect the amount of periodicity that a circuit exhibits. This addition will need to be made to help speed the process of reducing non-periodic functions.

References:

[1] Akers, S. B., Binary Decision Diagrams, *IEEE Transactions on Computers, vol.* C-27, no. 6, pp. 509-516, June 1978.

[2] Lee, C. Y., Representation of Switching Circuits by Binary-Decision Programs, *Bell System Technical Journal*, vol. 38, pp. 985-999, July 1959.

[3] Bryant, R. E., Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers*, vol. c-35, no. 8, pp. 677-691, August 1986.

[4] Bryant, R., Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, *ACM Computing Surveys*, vol. 24, no. 3, September 1992.

[5] Minato, S., Ishiura, N., Yajima, S., Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation, *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pp. 52-57, 1990.

[6] Minato, S., Graph-Based Representations of Discrete Functions, in **Representations of Discrete Functions**, edited by Tsutomu Sasao and Masahiro Fujita, Kluwer Academic Publishers, Boston, Massachusetts, 1996.

[7] Bollig, B., Wegener, I., Improving the Variable Ordering of OBDDs is NP-Complete, *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 993-1002, September 1996.

[8] Moore, R. P., Probability Based Variable Ordering and Reordering Heuristics for Decision Diagrams, M.S.C.S.E. thesis, Department of Computer Systems Engineering, University of Arkansas, August, 1997.

[9] Fujita, M., Fujisawa, H., Kawato, N., Evaluation and Improvement of Boolean Comparison Method based on Binary Decision Diagrams, *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pp. 2-5, November 1988.

[10]Rudell, R., Dynamic Variable Ordering for Ordered Binary Decision Diagrams, *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pp. 42-47, 1993.

[11]Bollig, B., Lobbing, M., Wegener, I., Simulated Annealing to Improve Variable Orderings for OBDDs, *Proceedings of the ACM/IEEE International Workshop on Logic Synthesis*, pp. 5-1 – 5-10, 1995.

[12]Thornton, M. A., Williams, J. P., Drechsler, R., Drechsler, N., Variable Reordering for Shared Binary Decision Diagrams Using Output Probabilities, *Proceedings of the Conference on Design, Automation and Test in Europe*, March 1999 (to appear).

[13]Parker, P., McClusky, E. J., Probabilistic Treatment of General Combinational Networks, *IEEE Transactions on Computers*, vol. c-24, pp. 668-670, June 1975.

[14] Somenzi, F., http://bessie.colorado.edu/~fabio/ , 1997.

[15]Drechsler, R., Becker, B., Gockel, N., A Genetic Algorithm for Variable Ordering of OBDDs, *IEE Proceedings*, 143(6):364-368, 1996.

[16]Hu, A. J., Formal Hardware Verification with BDDs: An Introduction, *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 677-682, 1997.

[17]Gupta, A., Formal Hardware Verification Methods: A Survey, *Formal Methods in System Design*, vol.1, no. 2/3, pp. 152-238, 1992.

[18]Thornton, M. A., Nair, V.S.S, Behavioral Synthesis of Combinational Logic using Spectral Based Heuristics, *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 2, April, 2000.

[19]Thornton, M. A., Nair, V.S.S., Efficient Calculation of Spectral Coefficients and Their Applications, *IEEE Transactions on Computer Aided Design*, vol. 14, no. 11, pp. 1328-1341, November, 1995.

[20]Ghosh, A., Devadas, S., Keutzer, K., White, J., Estimation of Average Switching Activity in Combinational and Sequential Circuits, *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 253-259, June, 1992.

[21]Savir, J., Ditlow, G., Bardell, P., Random Pattern Testability, *IEEE Transactions on Computers*, vol. C-33, no. 1, pp. 1041-1045, January, 1984.

[22]Critic, M., Estimating Dynamic Power Consumption of CMOS Circuits, *Proceedings of the International Conference on Computer-Aided Design*, pp. 534-537, 1987.

[23]Goldstein, H., Controllability / Observability of Digital Circuits, *IEEE Transactions on Circuits and Systems*, vol. 26, no. 9, pp. 685-693, September, 1979.

[24]Seth, S., Pan, L., Agrawal, V., PREDICT - Probabilistic Estimation of Digital Circuit Testability, *Proceedings of the Fault Tolerant Computer Symposium*, pp. 220-225, 1985.

[25]Krishnamurthy, B., Tollis, I., Improved Techniques for Estimating Signal Probabilities, *IEEE Transactions on Computers*, vol. 38, no. 7, pp. 1041-1045, July, 1989.

[26]Panda, S., Somenzi, F., Who Are the Variables in Your Neighborhood, *Proceedings of the IEEE/ACM Workshop on Logic Synthesis*, pp. 5-19, May, 1995.

[27]Fujita, M., Matsunaga, Y., Kakuda, T., On Variable Ordering of Binary Decision Diagrams for Applications of Multi-level Synthesis, *Proceedings of the European Conference on Design Automation*, pp. 50-54, 1991.

[28]Moller, D., Molitor, P., Drechsler, R., Symmetry Based Variable Ordering for ROBDDs, *IFIP Workshop on Logic and Architecture Synthesis*, pp. 47-53, 1994.

[29]Panda, S., Somenzi, F., Plesier, B. F., Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams, *Proceedings of the International Conference on Computer Aided Design*, pp. 628-631, November, 1994.

[30]Minato, S., Minimum-Width Method of Variable Ordering for Binary Decision Diagrams, *IEICE Transactions on Fundamentals*, vol. E75-A, no. 3:392-399, 1992.

[31]Malik, S., Wang, A. R., Brayton, R. K., Sangiovanni-Vincentelli, A., Logic Verification suing Binary Decision Diagrams in a Logic Synthesis Environment, *Proceedings of ICCAD*, pp. 6-9, 1988.