BDD-BASED VERIFICATION OF SCALABLE DESIGNS

Daniel Große and Rolf Drechsler
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{grosse,drechsle}@informatik.uni-bremen.de

Abstract

Many formal verification techniques make use of *Binary Decision Diagrams* (BDDs). In most applications the choice of the variable ordering is crucial for the performance of the verification algorithm. Usually BDDs operate on the Boolean level, i.e. BDDs are a bit-level data structure.

In this paper we present a method to speed-up BDD-based verification of scalable designs that makes use of a learning process for word-level information. In a pre-processing a scalable ordering is extracted from the RTL that is used as a static ordering for large designs. Experimental results show that significant improvements can be achieved.

Introduction

As modern circuits contain up to several million transistors, verification has become the major bottleneck in the design flow, i.e. up to 80% of the overall design costs are due to verification. This is one of the reasons why recently several formal verification methods have been developed since classical simulation cannot guarantee sufficient coverage of the design. E.g. in [1] it has been reported that for the verification of the Pentium IV more than 200 billion cycles have been simulated, but this only corresponds to 2 CPU minutes, if the chip is run at 1 GHz.

As alternatives, formal verification or symbolic simulation have been proposed and in the meantime these have been successfully applied in many projects [5]. In this context many alternative techniques have been proposed that are used to speed up the proof process, such as SAT or BDDs. A lot of work has been done to combine these techniques resulting in very efficient solvers (see e.g. [7]). Even though these techniques are very powerful they all operate on the Boolean level, i.e. high-level information that is available on the initial RTL description is not used. This also applies in cases where very regular structures are verified, such as adders, multipliers or scalable designs. Many difficulties in the proof process result from the fact that this information is not used. In contrast, the frontends that read in the RTL – typically given as Verilog or VHDL – transform the design to a flat netlist that only consists of AND-gates and inverters. This has shown several advantages for verification tools, but all structural information gets lost.

The major problem when using BDDs in the verification process is that a good variable ordering has to be determined. But this is an NP-complete problem and thus heuristics have to be applied. The most promising approaches regarding quality of the BDD are based on dynamic reordering of variables, like sifting [10]. Even though the resulting BDDs are small in size, the run times are prohibitive large, such that sifting is usually switched off during BDD construction. Alternatively, static variable ordering methods have been proposed that compute a BDD from the circuit topology (see e.g. [6]). But these approaches often fail to determine good results. All techniques proposed so far do not make use of high-level information or consider the scalability of the design.

In this paper we present a new technique to speed up BDD-based formal verification of scalable designs. In a first step a small instance of the *Device Under Verification* (DUV) is generated and the corresponding BDD is build. This BDD is optimized based on dynamic variable reordering. Since the instance is small, this process runs very fast. Then the resulting optimized variable ordering is analyzed using a pattern matching approach. After this phase the ordering is scaled based on word-level information extracted from the signal names. This scaled ordering is then used as a static ordering for larger instances.

Experimental results for verification of combinational and sequential circuits showed significant reductions, i.e. instances that took several hours before could be verified within a few seconds.

The paper is structured as follows: First we introduce basic definitions. Then we give the main idea of the approach. In the following section our approach is discussed in detail. Next the experimental results are presented. Finally, the work is summarized.

Preliminaries

As is well-known a Boolean function $f : B^n \rightarrow B$ can be represented by a *Binary Decision Diagram* (BDD) which is a directed acyclic graph where a Shannon decomposition

$$f = \overline{x_i} f_{x_i=0} + x_i f_{x_i=1} \ (1 \leq i \leq n)$$

is carried out in each node. A BDD is called *ordered* if each variable is encountered at most once on each path from the root to a terminal node and if the variables are encountered

in the same order on all such paths. A BDD is called *reduced* if it does not contain isomorphic sub-graphs nor does it have redundant nodes. Reduced and ordered BDDs are a canonical representation since for each Boolean function the BDD is uniquely specified [2]. In the following, we refer to reduced and ordered BDDs for brevity as BDDs. It is well known that BDDs are very sensitive to the chosen variable ordering, i.e. the size may vary from linear to exponential.

Basic Idea

Before the algorithm is described in detail, the underlying main idea and the resulting four steps of our technique are first illustrated by a simple example:

Consider the *n*-bit adder with operands *a* and *b*, where a_0 and b_0 denote the least significant bit, respectively. It is known for adders that an interleaved order gives an optimal result, if the bits are ordered from the least to the most significant bit, i.e.:

$a_0, b_0, a_1, b_1, a_2, b_2, \ldots, a_{n-1}, b_{n-1}$

In this case the resulting BDD has linear size. But if the operands are separated, like

$a_0, a_1, a_2, a_3, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{n-1},$

the resulting BDD has exponential size.

The proposed technique works in four steps:

1. Build the BDD for a small number of bits only.
2. Perform an optimization based on dynamic reordering.
3. Analyze the ordering and generalize it to an *n*-bit order.
4. Build the BDD for the large number of bits based on a static ordering.

In the example we start with the "worst case" ordering, i.e. for the adder this means that the two operands *a* and *b* are separated. If we start with a small number of bits, e.g. 10 bits, then sifting determines an interleaved ordering that is afterwards generalized and used as a static ordering for building a 32-bit adder.

The benefit of this approach is obvious: Since the time consuming Step 2 of BDD minimization is only carried out on a small design with a small number of variables, the algorithm runs very fast and due to the regularity of the design the quality is very good as will be shown by experiments later.

Even though the method is simple regarding the general approach, it has shown to be very effective. In the following we first describe the analysis phase in more detail and then discuss case studies of scalable designs. It is shown that speed-ups of several orders of magnitude can be achieved.

Scaling BDD Ordering

While the processing in Steps 1, 2 and 4 in the previous section are rather obvious, the crucial step in the approach is the analysis phase. Based on the ordering for the small example the ordering for the *n*-bit version is extrapolated. The approach would of course benefit from various runs, i.e. if several orders could be considered. This results from the fact that sifting is also a heuristic approach and by several runs robustness can be obtained. In the following only a single variable ordering is studied, since our experiments have shown that this is sufficient. But, it should be noticed that this might become necessary for more complex and more irregular designs.

The resulting ordering is considered as a string of characters, where in each position the name of the corresponding input is given. In the example above this would correspond to e.g. a_0 or b_5. The text string is evaluated by determining the relative order of each entry. This is then matched against existing patterns. From our studies and assuming regularity in a scalable design, it turned out that it is sufficient to consider only four patterns:

1. Increasing
2. Decreasing
3. Interleaved increasing
4. Interleaved decreasing

In the case of the adder above, this corresponds to:

1. $a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{n-1}$
2. $a_{n-1}, \ldots, a_1, a_0, b_{n-1}, \ldots, b_1, b_0$
3. $a_0, b_0, a_1, b_1, \ldots, a_{n-1}, b_{n-1}$
4. $a_{n-1}, b_{n-1}, \ldots, a_1, b_1, a_0, b_0$

If blocks are more complex, i.e. they do not consider a single bit as in the case of the adder, the method has also to take this hierarchy into account. Notice that the approach not only works for combinational but also for sequential circuits. In this case also variables for the present states and next states are part of the BDDs but they can be treated in the same way. The next state variables are necessary for computing the transition relation of the sequential circuit.

In the following the analysis phase is described in more detail.

Analysis of Ordering

Given a scalable design consisting of *n* blocks. Then the corresponding BDD ordering string is of the form "$a_i\ b_i\ c_i\ d_i \ldots$" where *i* is the number of a block and each character string corresponds to an input, a current state or a next state variable of a block. The current state and next state variables are used for representing the transition relation. The ordering analysis algorithm is split into two parts. The first part is used to identify increasing or decreasing patterns. The second part is applied to identify the interleaved increasing or decreasing patterns.

A sketch of the analysis algorithm is given in Figure 1. The first part of the algorithm works as follows (for the

integrated examples assume that the given ordering string *os* is "$a_0\ a_1\ a_2\ a_3\ c_0\ c_1\ c_2\ b_3\ c_3\ b_0\ b_1\ b_2$"):

1. First the number *nv* of different variable names and the different variable names are determined. (Example: *nv* is *3* and the variable names are *a*, *c* and *b*).
2. Now the variables with the same names are collected and consecutive variables in the ordering string are enclosed by brackets. This results in *nv* strings. (Example: "$[a_0\ a_1\ a_2\ a_3]$", "$[c_0\ c_1\ c_2]\ c_3$" and "$b_3\ [b_0\ b_1\ b_2]$").
3. For each string of Step 2 the longest consecutive string is considered and increasing or decreasing of indices is measured. This is realized by comparing the index *i* of a variable with the index *j* of each successor. If $i < j$ then an increasing pair is found, if $i > j$ the pair is decreasing. In order to obtain an overall score for increasing/decreasing of all longest consecutive strings, the number of all increasing/decreasing pairs is counted. (Example: increasing is $6+3+3=12$ and decreasing is $0+0+0=0$).
4. Now by starting from the left side of the ordering string the run lengths of consecutive variable names are counted. From this result the maximum run length of each variable name is accumulated to *maxf* and a list *relativeOrderList* with the corresponding variable names is generated. (Example: $4 \cdot a$, $3 \cdot c$, $1 \cdot b$, $1 \cdot c$, $3 \cdot b$, *maxf* is *10* and *relativeOrderList = a c b*).
5. Then the ratio *maxf / (number of variables)* is computed. This ratio indicates the probability of an increasing or decreasing pattern. (Example: *ratio* is *10/12 = 0.83*).
6. If ($ratio \geq 0.75$) then the ordering string is an increasing or decreasing pattern. In this case the overall result of the first part of the analysis algorithm is increasing or decreasing depending on a comparison of increasing and decreasing from Step 3 and the *relativeOrderList* from Step 4. (Example: increasing, because $12 > 0$ and "*a c b*", i.e. scaled ordering for *n* will be "$a_0 \ldots a_{n-1}\ c_0 \ldots c_{n-1}\ b_0 \ldots b_{n-1}$").

Notice that the described first part of the analysis algorithm does not find a solution for interleaved increasing/decreasing orderings. So to identify this type of orderings the following pattern matching technique is applied (assuming ordering string *os* to be "$a_0\ b_0\ c_0\ a_1\ c_1\ b_1\ a_2\ b_2\ c_2\ a_3\ b_3\ c_3$"*):*

1. First it is determined whether the total ordering is mostly increasing or mostly decreasing. This works by comparing the index of a variable with all the indices of its successor variables analogously to the third step of the first part of the analysis algorithm. (Example: *mostly_increasing* is $9 \cdot 3+6 \cdot 3+3 \cdot 3=54$ and *mostly_decreasing* is *0*).
2. In the second step the ordering string is scanned from the beginning and for each block all variables are collected. During this scanning also all consecutive variables of the same block are enclosed by brackets. This results in *n* relative ordering strings ro_i each containing all variables of the corresponding block *i*. The goal of this step is to determine the relative order within each block. (Example: ro_0="$[a_0\ b_0\ c_0]$", ro_1="$[a_1\ c_1\ b_1]$", ro_2="$[a_2\ b_2\ c_2]$", ro_3="$[a_3\ b_3\ c_3]$").
3. Then starting with the longest consecutive string of block *0*, this string is matched against all other strings of the same length of the following blocks. This matching works only considering the names of variables, i.e. for example the string $[a_0\ b_0\ c_0]$ of length *3* from block *0* matches the string $[a_2\ b_2\ c_2]$ of block *2*. The number of matches for every string is counted. This method is iterated for all consecutive strings down to length *2*. As a result, every consecutive string obtains a score determined by $string\_length \cdot matches$. (Example: highest score is $6=3 \cdot 2$ of ro_0).
4. If the ordering string is not as regular as in the example the string *sh* with the highest score does not contain all variables of a block, i.e. *sh* is only a sub-string of some ro_i. So all ro_i which contain *sh* (only variable names are matched) are compared with all following ro_j analogously to Step 3. The most frequently matched ro_i represents the local ordering of a block and will be used as a result together with increasing or decreasing based on decision of Step 1. (Example: increasing and "$a_0\ b_0\ c_0$", i.e. scaled ordering for *n* will be "$a_0\ b_0\ c_0\ a_1\ b_1\ c_1 \ldots a_{n-1}\ b_{n-1}\ c_{n-1}$").

With the described analysis algorithm the ordering of a small instance can be analyzed and a generalization for larger designs can be computed.

In the following experimental results show the efficiency of the approach.

Experimental Results

In this section experimental results are given. The proposed technique has been implemented in C++. All run times are given in CPU seconds on an Intel Pentium IV with 1,7 GHz and 512 MByte of main memory. As the BDD package we used CUDD [11]. The run times given for our approach always contain the times for the complete flow, i.e. including analysis and construction for small instances. For the experiments three scaleable designs have been considered:

1. Adders
2. Multipliers
3. Arbiters

```
scale_ordering(ordering_string os) {
  // first part: identify increasing/decreasing
  nv = get number of different variable names (os); varNames = get variable names (os);
  equalVarNamesList = collect equal variable names and enclose consecutive variables (varNames, nv);
  increasing = decreasing = 0;
  for (each s in equalVarNamesList) {
      ls = get longest enclosed string(s);
      for (each variable v_i in ls) {
          for (each variable v_j after v_i in ls) {
              if (i < j) increasing++;
              if (i > j) decreasing++;
          }
      }
  }
  runLengthList = count run lengths of consecutive variable names (os);
  maxf = accumulate maximum run length of each variable name (runLengthList);
  relativeOrderList = maximum frequently variables (runLengthList);
  ratio = maxf / (number of variables);
  if (ratio ≥ 0.75) {
      if (increasing > decreasing) return (increasing, relativeOrderList);
      else return (decreasing, relativeOrderList);
  }
  // second part: identify interleaved increasing/decreasing
  mostly_increasing = mostly_decreasing = 0;
  for (each variable v_i in os) {
      for (each variable w_j after v_i in os) {
          if (i < j) mostly_increasing++;
          if (i > j) mostly_decreasing++;
      }
  }
  roList = collect variables belonging to the same block and enclose consecutive variables (os);
  roLengthList = get different lengths of consecutive variable strings (roList);
  for (decreasing length l in roLengthList) {
      roSameLengthList = get all consecutive variable strings with length l (roList);
      for (each s in roSameLengthList) {
          for (each t in roSameLengthList after s in roSameLengthList) {
              if (s matches t) matches++;
          }
          score[s] = l · matches;
      }
  }
  sh = get string with highest score (score);
  if (sh does not contain all variables of a block) {
      roContainShList = get all ro_i where sh matches (roList);
      compare each s in roContainShList with all t in roContainShList after s;
      sh = most frequently matched string of comparison;
  }
  if (mostly_increasing > mostly_decreasing) return (interleaved increasing, sh);
  else return (interleaved decreasing, sh);
}
```

Figure 1. Sketch of ordering analysis algorithm

While the first two are combinational instances, the third class describes a sequential problem, i.e. the computation of reachable states.

The first two instances are very different in nature, since adders are known to be very easy to verify by BDDs, if a good variable ordering is chosen. But BDDs always blow up for multipliers. Our experiments will demonstrate that the approach has significant advantages in both cases: For adders the construction is sped up significantly for larger instances, while the method also has benefits for difficult instances, like multipliers. In this case the method gives up very fast, while classical approaches, like sifting, spend a lot of time on useless optimization runs.

Adders

The results for the adder circuits are given in Table 1. In the first column the number of bits to be added are given. Then for both approaches *Memory* and *Time* denote the memory in MByte used by the BDD manager and the run time in CPU seconds, respectively. A time limit for BDD construction of 2 CPU hours has been set. As can be seen, already for 20 variables, the new approach outperforms sifting. For 500 variables, the scaling technique is nearly a factor of 10 faster.

Table 1. Results for adders

Bits	Sifting		Scaling	
	Memory	Time	Memory	Time
10	4.62	0.01	4.64	0.09
20	4.64	0.05	4.64	0.09
30	4.66	0.08	4.64	0.10
40	4.68	0.15	4.64	0.12
50	4.71	0.26	4.64	0.13
60	4.73	0.34	4.64	0.16
70	4.75	0.48	4.67	0.19
80	4.77	0.62	4.69	0.22
90	4.79	0.90	4.71	0.25
100	4.81	1.12	4.73	0.30
200	5.02	7.64	4.97	1.54
300	5.22	23.94	5.19	5.05
400	5.43	55.99	5.43	10.05
500	5.69	114.13	5.65	15.83
600	-	-	5.89	22.96
700	-	-	6.12	31.49
800	-	-	6.36	40.85
900	-	-	6.59	51.67
1000	-	-	6.82	64.46
1100	-	-	11.06	77.40
1200	-	-	11.28	92.41
1300	-	-	11.52	108.64
1400	-	-	11.74	126.10
1500	-	-	11.98	144.39

Multipliers

In a next series of experiments we consider multiplier circuits. It is well known that BDDs always become exponential in the number of variables independent of the chosen variable ordering [3]. For this, it is interesting to study the run time of the algorithms until they give up. We started with a live node limit of 2,000,000 BDD nodes. For up to 12-bit multipliers the BDDs can be constructed. For larger instances the construction failed (shown in italic). We report the memory consumption and the run time for sifting and our approach until 12-bit. Beyond 12-bit the memory and run time used until the construction failed is given. In case of sifting the values are not monotonically increasing because sifting is called dynamically by the BDD package. Since, in the final phase of our approach a static variable ordering is applied, the limit is reached very fast, as can be seen in Table 2. Compared to sifting a speed-up of more than a factor of 20 can be observed for a 12-bit multiplier.

Table 2. Results for multipliers

Bits	Sifting		Scaling	
	Memory	Time	Memory	Time
5	4.55	0.04	5.44	0.87
6	4.66	0.10	5.44	0.92
7	4.78	0.26	5.44	1.03
8	5.44	0.81	5.90	1.29
9	6.33	6.47	8.82	1.96
10	13.52	18.26	29.89	3.89
11	30.26	101.69	57.48	11.08
12	53.03	721.77	59.65	35.00
13	*68.17*	*1047.09*	*58.21*	*23.45*
14	*76.59*	*1452.54*	*61.70*	*37.87*
15	*73.31*	*1329.90*	*63.14*	*46.91*
16	*65.25*	*808.06*	*62.89*	*46.44*
17	*74.51*	*1362.95*	*60.03*	*54.56*
18	*55.31*	*538.30*	*65.52*	*63.07*
19	*70.77*	*1018.71*	*60.06*	*66.78*
20	*55.81*	*604.59*	*60.40*	*73.02*

Arbiters

As a sequential benchmark for our experiments we considered a scalable bus arbiter. This circuit is often used for experiments in formal verification (see e.g. [8,9]). In the upper part of Figure 2 a single arbiter cell is shown, whereas the composition to an *n*-cell arbiter is given in the lower part.

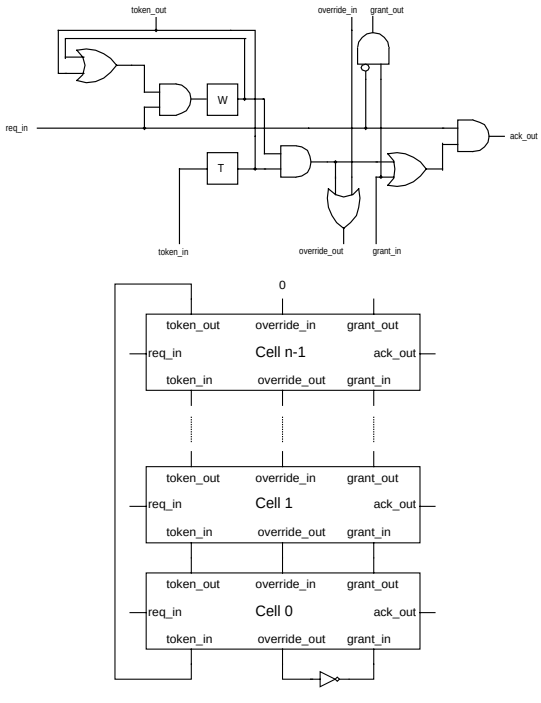

Figure 2. A scalable bus arbiter

For the resulting circuit a computation of the reachable states is carried out. For the new approach the analysis phase was run on an example with 20 cells. The run times are negligible, since also sifting for these

instances needs nearly no time. In the following we give the results for a complete reachability analysis using sifting and the scaling approach. The results are given in Table 3. In the first column the number of arbiter cells is given. The second column shows the overall number of BDD variables. Then as above for both approaches memory and time is given.

As has been shown in [4] the reachability analysis can be performed up to n=11 bits with 512MB of memory, if the original variable ordering as it occurs in the benchmark description is used and sifting is disabled. With sifting this can be improved. But already for 300 cells more than 7200 CPU seconds (corresponding to 2 CPU hours) are needed. The arbiter with 200 cells already takes more than 3000 CPU seconds, while the scaling approach can handle this instance - including the pre-processing - within 5 seconds, i.e. a speed-up of more than a factor of 600. Using the new technique the complete reachability can be computed for up to 1500 arbiter cells in about 1000 CPU seconds.

Conclusions

A new approach for finding BDD orderings has been proposed. This technique works for scalable designs and makes use of high-level information. Experimental results have demonstrated the quality of the approach. In contrast to dynamic reordering improvements of several orders of magnitude have been observed.

It is focus of current work to integrate the approach in an existing verification flow [5]. Here it is important that the ordering can be given to the tool without changing any of the internal structures, but in the form of a pre-processing.

Table 3. Results for scalable arbiter

Cells	BDD	Sifting		Scaling	
	Variables	Memory	Time	Memory	Time
100	500	13.37	195.93	8.05	1.18
200	1000	39.91	3126.84	31.93	4.55
300	1500	-	-	37.75	12.79
400	2000	-	-	48.73	28.25
500	2500	-	-	47.29	49.45
600	3000	-	-	54.27	81.65
700	3500	-	-	57.32	122.31
800	4000	-	-	57.74	176.23
900	4500	-	-	61.63	238.55
1000	5000	-	-	66.10	320.48
1100	5500	-	-	67.02	412.10
1200	6000	-	-	72.92	540.57
1300	6500	-	-	79.79	670.39
1400	7000	-	-	87.47	822.19
1500	7500	-	-	100.23	1006.89

References

[1] B. Bentley. Validating the Intel Pentium 4 microprocessor. In Design Automation Conf., pp. 244-248, 2001.

[2] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. IEEE Trans. On Comp., 35(8):677-691, 1986.

[3] R.E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. IEEE Trans. On Comp., 40:205-213, 1991

[4] R. Drechsler and D. Große. Reachability Analysis for Formal Verification of SystemC, EUROMICRO Symposium on Digital System Design (DSD'2002), Dortmund, pp. 337-340, 2002

[5] R. Drechsler, S. Höreth, Gatecomp: Equivalence Checking of Digital Circuits in an Industrial Environment, International Workshop on Boolean Problems, pp. 195-200, 2002

[6] H. Fujii, G. Ootomo, C. Hori, Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams, ICCAD, pp. 38-41, 1993

[7] A. Kuehlmann, M. Ganai, V. Paruthi, Circuit-Based Boolean Reasoning, In Design Automation Conference, pp. 232-237, 2001

[8] K.L. McMillan. Symbolic Model Checking. Kluwer Academic Publisher, 1993.

[9] J. Ruf, D. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi valued AR-automata. In Design, Automation and Test in Europe, pp. 742-748, 2001.

[10] R. Rudell, Dynamic Variable Ordering for Ordered Binary Decision Diagrams, ICCAD, pp. 42-47, 1993

[11] F. Somenzi. CUDD: CU Decision Diagram Package Release 2.3.1, University of Colorado at Boulder, 2001.