Forschungsbericht Nr. 98-24

Application Driven Variable Reordering and an Example Implementation in Reachability Analysis

Christoph Meinel   Klaus Schwettmann   Anna Slobodová

This work was partially supported by DFG grant Me 1077/12-1 and Graduierten Kolleg "Mathematische Optimierung"

Application Driven Variable Reordering and an Example Implementation in Reachability Analysis

Christoph Meinel[†,‡] Klaus Schwettmann[‡] Anna Slobodová[†]

Abstract

Variable reordering is the main approach to minimize the size of Ordered Binary Decision Diagrams. But despite the huge effort spent, up to now, to design different reordering heuristics, their performance often does not meet the needs of the applications. In many OBDD-based computations, the time cost for reordering dominates the time spent by the computation itself.

There are some known approaches for accelerating the reordering by taking advantage of structural properties of OBDDs and functions represented. In this paper, we propose a reordering method that exploits application specific information. The main idea is to drive the reordering process by the computation. This effects an acceleration of the whole computation rather than of the reordering only. The power of the approach is illustrated on hand of speeding up forward traversal of finite state machines.

1 Introduction

Boolean operations, tautology, satisfiability and equivalence tests, can be performed efficiently by means of Ordered Binary Decision Diagrams (OBDDs[Bry86]) as long as their size remains manageable. Hence, the key problem is to keep the OBDDs as small as possible during the entire computation. Although the OBDD representation is canonical for any fixed variable order, different variable orders yield not only different representations, but representations with possibly an exponential difference in the size. Therefore, the choice of an appropriate variable order appears indispensable for any manipulation with OBDDs.

Because of intractability of finding an optimal variable order, which is proved to be an NP-complete problem [THY93], a series of heuristics has been developed for deriving an initial order from the topology of a circuit or a formula description (e.g., [MW$^+$88]), and for its dynamic change (e.g., [Rud93, FY$^+$93, PSP94, DBG96, MS97]). Dynamic reordering techniques are usually based on a greedy method (with respect to the size of OBDDs) that changes a current variable order by means of local or global modifications. One of the most successful minimization techniques is Sifting, proposed by Rudell [Rud93]. Sifting is a general purpose reordering technique that is based on exchange of two neighbouring variables – the so-called *swap* operation. Each variable is moved through the whole order by means of swaps and placed on the position that minimizes the size of the OBDDs. The algorithm blindly searches for a better variable order without using any additional information from the particular application the OBDDs are used for.

Attempts to control sifting by use of some additional information about the structure of the involved OBDDs and some particular properties of the represented functions have already been done, e.g., in [PSP94, PS95, MS97]. In this paper, we take a further step in this direction by using application specific information. Our approach - an application driven reordering – is aimed to the acceleration of the whole OBDD-based computation rather than to speeding up the reordering only. It is built on the idea of sample reordering introduced in [SM98]. The power of the method is demonstrated by means of the reachability analysis of finite state machines, in particular on forward traversal. However, the approach can be adapt to any other computation. Experimental results are obtained on benchmark circuits using VIS [VIS96] with the CUDD package [Som97] embedded.

2 ADR – Application Driven Reordering

OBDD based computation is a dynamic process: New OBDDs are created from existing ones by means of composition or other operations, and the OBDDs that become unnecessary for further use by the application are discarded. Since the most critical resource is the memory, freeing useless OBDDs and dynamic garbage collection is an important part of solving real world problems. However, even the best garbage collection strategy appears insufficient for a complete solution of the memory problems. Since the set of represented functions changes

*TI, Bahnhofstr. 30-32, 54292 Trier, Germany. {meinel, slobodova}@ti.fhg.de

†University of Trier, FB IV–Informatik, 54286 Trier, Germany. {meinel, schwettm, slobodova}@ti.uni-trier.de

over time, it is inefficient to work with a fixed variable order that is chosen heuristically at the beginning. The second problem is that even if there is a variable order that is good for all OBDDs that are built (and eventually discarded) during the computations, the known heuristics do not necessarily find it. The standard method of keeping OBDDs small is to allow a dynamic minimization that invokes a reordering heuristic whenever the size of the OBDDs grows too much.

The quality of a heuristic is usually measured by the size of the obtained OBDDs. But it turns out that in more complex computations, the portion of the time spent by reordering is too large with respect to the time spent by problem solving itself. Hence, the time cost of a heuristic has the highest priority in our evaluation as long as the achieved reduction of OBDD sizes is satisfiable.

Another point to be mentioned is the following one. In a large OBDD-based application, there may occur many OBDDs that will not be active in the next step, but they cannot be removed because they will be used later. On the other hand, there are OBDDs (that may be only a small portion of all OBDDs) that take actively part in the next operations. A blind reordering that does not make any difference between these two different OBDD categories will look for an order that minimizes the whole OBDDs as much as possible. Our idea is to give some OBDDs a higher weight for minimization. The goal is to find a good order for the specified set of OBDDs that simultaneously keeps the entire OBDDs as small as possible. This reduces the required CPU time, since, due to the approach, the *active* OBDDs are smaller.

Taking into account the arguments mentioned above, and combining them with the idea of Sampling Reordering proposed in [SM98], we suggest the following method called *Application Driven Reordering (ADR)*: Analyzing the meaning of functions represented by the OBDDs in question in the context of a particular application, we determine which OBDDs will be *active* and/or *important* in the next computational step. *Active* means that they will be modified or used for the creation of new OBDDs. The *next step* of the computation may mean one or more individual operations on OBDDs. The active OBDDs can be divided into several groups with assigned weights. A sample of OBDDs is chosen from the OBDDs in the groups. The size of the sample is a small portion of the size of the entire OBDDs, and the size of the OBDDs that represent a group in the sample is proportional to the group weight, which is a parameter of the method. At first, the sample is reordered by means of any heuristic. Reordering of the sample takes much less time than the reordering of the entire OBDDs because of the small sample size. Then the variable order obtained by minimization of the sample is extrapolated and applied to the entire OBDDs by a procedure *ChangeOrder* described in Section 2.1.

If the reduction of the OBDD size reaches the value of a parameter *ExpectedReduction*, then the reordering is considered as successfully finished and we transfer control back to the application. Otherwise, a second attempt is performed with a new sample. In order to avoid the choice of the same sample twice, we introduce some randomness.

Application Driven Reordering can be implemented in any general purpose OBDD package as a reordering option. The actual choice of the sample is a subject of an application. A list of OBDDs that should be preferably chosen into the sample is passed from a higher level application software to the OBDD package. A random sample is chosen, if the list is empty. This scheme requires a negligible extension of an application.

2.1 ASR applied to Forward Traversal of FSMs

Let us consider an implementation of the idea described above within the reachability analysis of Finite State Machines.

Definition: *A Finite State Machine* (FSM) is a 6-Tuple $(\Sigma, O, S, S^0, \delta, \lambda)$, where Σ is an input alphabet, O an output alphabet, S a finite state set, $S^0 \subset S$ the set of initial states, $\delta : S \times \Sigma \to S$ a transition relation, and $\lambda : S \times \Sigma$ an output function.

A transition relation δ describes the change of a current state according to the current input. A state t is reachable from a state s in one step if there is an input x such that $t = \delta(s, x)$. t is reachable from s if for some $i \geq 0$ there is a sequence of states $s_0, s_1, \ldots, s_i$ and a sequence of inputs $x_0, x_1, \ldots, x_{i-1}$ such that $s_0 = s, s_i = t$, and for each $0 \leq j < i$ it holds: $s_{j+1} \in \delta(s_j, x_j)$.

The basic task in manipulation of FSMs is the computation of the set of states that are reachable from the initial states. One of the standard techniques for computing this set is the *Forward Traversal* (Fig. 1) based on a breadth-first-search fixpoint iteration, see e.g., [CBM89, HS96] for details.

The state and the input set of the FSMs used in practical applications are too large to be handled explicitly. Hence, the states and the inputs are encoded by a sequence of Boolean variables. This allows symbolic manipulation with state sets. In the case of the FSM traversal, the sets S^0, *To, From, New*, and *Reached* are represented implicitly by the OBDDs of their characteristic functions. *Image*(δ, `From`) computes those states that are reachable from the states in *From* within one step.

```
Forward Traversal(δ, S⁰) {
   Reached=From=S⁰;
   do{
        To=Image(δ,From);
        New=To \ Reached;
        From=New;
        Reached=Reached∪New;
   } while(New≠ ∅);
   return Reached;
}
```

Figure 1: Forward Traversal.

If the FSM traversal appears as part of a more complex application, e.g., equivalence check of two automata, the OBDDs that are used in the computation of *Reached* may be a small but nonetheless an important part of the entire OBDDs. For the proposed ADR, we chose a sample from four groups of OBDDs:

1. The first group consists of the OBDDs that represent the transition relation. This can be one OBDD (in the case of monolithic representation) or several (in the case of clustered OBDDs) e.g.,[RA+95]. The OBDDs in this group are not changed at all during the traversal.
2. The second group contains only the characteristic function of the set *Reached*.
3. The third group contains the characteristic function of the set *To*.
4. The last group contains those OBDDs that have been currently computed and do not appear in any of the previous groups. They are computed as a difference between the OBDDs that exist at the moment of the particular invocation of the reordering, and those OBDDs that were present at the beginning of the current iteration step or are already included in the groups 1-3.

Since the OBDDs in all but the first group are changed during the traversal, they are dynamically updated. The weights of the groups are set proportionally to the size of the OBDDs in a group compared to the size of the entire OBDDs. More precisely, let *GroupSize* be the number of nodes in the OBDDs of a group, *OBDDsize* be the shared size of all OBDDs, and $SampleSize = OBDDsize \times SamplePortion$ be the required size for the sample computed according to a parameter *SamplePortion*. Then the OBDDs in the group contribute by

$$SampleIncrease = \frac{GroupSize}{OBDDsize} \times \text{SampleSize}$$

nodes to the sample. The fourth group tends to be of special importance during the image computation and often contributes to more than a half of the sample. Because of the sharing of nodes among the groups, it may happen that it is not possible to add *SampleIncrease* nodes from the group. In this case, the missing nodes are chosen randomly from the entire OBDDs. Even if there are sufficiently large OBDDs in the group, there is a randomness in the choice of the sample. If there is only one OBDD in a group (e.g., groups 2 and 3), we choose an internal node by a random walk from the root and the subOBDD rooted in this node is added to the sample. If there are more roots in the group, we subsequently choose some of them such that their shared size covers *SampleIncrease*.

Once a sample is chosen, the algorithm proceeds as described above. The sample is minimized by any given heuristic (e.g., by Sifting). The new variable order of the sample is taken as a target order for rebuilding all OBDDs by using *ChangeOrder* (*ChangeOrder* is an extension of a shuffling procedure of the CUDD package with respect to variable groups). If this reordering yields the expected reduction,the heuristic terminates and the traversal algorithm may continue. Otherwise we do another attempt.

3 Experiments

This section is devoted to a first experimental evaluation of the Application Driven Reordering for *Forward Traversal* applied to some circuits of the LGSynth91 and ISCAS89 benchmarks. The reordering method was implemented in CUDD-2.1.2. [Som97] that is embedded in the VIS-1.2 package [VIS96]. All experiments ran on a Pentium Pro 200 with 512 MB memory. Our goal was to compare the proposed reordering method with Sifting, both used dynamically during the FSM traversal. The default methods "frontier" and "IWLS'95" [RA+95] were used for partitioning and forward traversal. During partitioning and the clustering phase of "IWLS'95", Sifting was used as dynamic reordering heuristic in all experiments. In the traversal phase of "IWLS'95", the experiments were split into experiments with dynamic Sifting and those with dynamic Application Driven Reordering. Parameters for Sifting were set according to the CUDD default settings. Parameters for ADR described in the previous section were set as follows: *ExpectedReduction*= 80% *SamplePortion*= 20% *ChangeOrderBound*= 140%

Table 1 contains the experimental results for a sample of circuits. The quality measure of a reordering method is the time required for traversal as long as the peak size (the maximal shared size of the OBDDs that are in memory at the same time) remains acceptable. The values in the column labeled by Time are given in seconds and amount the CPU time from the beginning of the traversal up to the point when the iteration step given by the column labeled by Depth is finished. In the most cases the traversal ran faster when Application Driven Reordering was used for dynamic reordering. The table shows results for the computation within two hours of CPU time, up to the last step which had been finished in this time (e.g. s1269 needs 13,435s for the third step with Sifting as reordering method). Not all computations could be completed within this time limit (the table entries filled by "—"). The Peak-Size is the maximum of the shared OBDD sizes, measured after each traversal step, up to the step given by column Depth.

References

[Bry86] R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transaction on Computers*, C-35:677–691, 1986.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of Synchronous Sequential Machines Based on Symbolic Execution. In *Automatic verification Methods for Finite State Systems*, pages 365–373. Springer LNCS 407, 1989.

[DBG96] R. Drechsler, B. Becker, and N. Göckel. A Genetic Algorithm for Variable Ordering of OBDDs. *IEEE Proc. Computers and Digital Techniques*, 143(6):364–368, 1996.

[FY+93] E. Felt, G. York, R. Brayton, and A. Sangiovanni-Vincentelli. Dynamic Variable Reoredering for BDD Minimization. *Proc. EURO-DAC*, pages 130–135, 1993.

[HS96] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[MS97] Ch. Meinel and A. Slobodová. Speeding up Variable Reordering of OBDDs. *IEEE Proc. of ICCD*, pages 338–343, 1997.

[MW+88] S. Malik, A. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. *IEEE Proc.of ICCAD*, pages 6–9, 1988.

[PS95] S. Panda and F. Somenzi. Who are the Variables in Your Neighbourhood. *Proc. ACM/IEEE ICCAD*, pages 74–77, 1995.

[PSP94] S. Panda, F. Somenzi, and B.F. Plessier. Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. *Proc. ACM/IEEE ICCAD*, pages 628–631, 1994.

[RA+95] R. K. Ranjan, A. Aziz, R. K. Braytonand, B. Plessier, and C. Pixley. Efficient BDD Algorithms for FSM Synthesis and Verification. In *Proceedings of the International Workshop on Logic Synthesis*, 1995.

[Rud93] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. *Proc. ACM/IEEE ICCAD*, pages 42–47, 1993.

[SM98] A. Slobodová and Ch. Meinel. Sample Method for Minimization of OBDDs. *Proc. SOFSEM'98*, to appear in Springer LNCS, 1998.

[Som97] F. Somenzi. CUDD: CU Decision Diagram Package. Release 2.1.2. *University of Colorado, Boulder(ftp://vlsi.colorado.edu/pub/)*, 1997.

[THY93] S. Tani, K. Hamaguchi, and S. Yajima. The Complexity of the Optimal Variable Ordering Problem of Shared Binary Decision Diagrams. In *Proc. ISAAC*, LNCS 762, pages 389–398. Springer, 1993.

[VIS96] The VIS Group. VIS: A System for Verification and Synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 428–432. ACM, Springer LNCS 1102, July 1996.

Traversaltime 2 hours							
		Sifting			ADR		
Circuit	Depth	Final-Size	Peak-Size	Time	Final-Size	Peak-Size	Time
s1269[c]	2	274,845	274,845	73	183,756	183,756	**55**
	3	——	——	——	281,439	281,439	**1,554**
s1423[a]	10	892,510	966,808	3,186	460,028	1,328,722	**1,108**
	11	——	——	——	1,383,183	1,383,183	**3,349**
s1512[c]	1023[d]	38,117	477,598	1269	47,680	494,478	**1057**
s3271[c]	8	872,205	872,205	2,023	287,476	686,012	**615**
s3384[c]	3	49,951	55,241	37	114,068	114,068	**23**
s4863[c]	4[d]	857,398	857,398	6,560	289,997	289,997	**1,927**
s5378[a]	3	183,406	183,406	**586**	1,125,997	1,125,051	793
	4	565,128	565,128	**2,078**	——	——	——
s9234.1[a]	8	76,446	637,829	**3,148**	——	——	——
s13207.1[b]	9	602,370	602,370	5,317	784,525	1,772,894	**4,093**
	10		——	——	931,975	1,772,894	**6,449**
s38584.1[a]	2	1,074,276	1,074,276	1,918	1,146,732	1,146,732	**591**
	3		——	——	2,568,209	2,568,209	**4,497**
dsip[a]	15[d]	127,121	171,755	**215**	49,059	594,333	729
mm9a [a]	3[d]	19,737	31,570	3	30,668	31,570	**2**

[a]LGSynth91
[b]ISCAS89
[c]ADDENDUM93
[d]traversal completed

Table 1: Experiments.