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) 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., [MW88]), 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 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 when-
ever 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 satisfi-
able.

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 cat-
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 pos-
able. 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 Re-
ordering 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 ap-
lication, we determine which OBDDs will be active
and/or important in the next computational step. Ac-
tive means that they will be modified or used for the
creation of new OBDDs. The next step of the computa-
tion may mean one or more individual operations on
OBDDs. The active OBDDs can be divided into se-
veral 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 sam-
ple 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 at-
tempt 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 prefer-
ably 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 re-
quires 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 \(\Sigma\) is an input alphabet,
\(O\) an output alphabet, \(S\) a finite state set, \(s^0 \subseteq S\) the
set of initial states, \(\delta : S \times \Sigma \rightarrow S\) a transition relation, and
\(\lambda : S \times \Sigma \rightarrow O\) an output function.

A transition relation \(\delta\) describes the change of a cur-
rent 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 com-
putation 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 prac-
tical applications are too large to be handled explicitly.
Hence, the states and the inputs are encoded by a se-
quence of Boolean variables. This allows symbolic ma-
nipulation with state sets. In the case of the FSM traver-
sal, the sets \(S^0\), \(To\), \(From\), \(New\), and \(Reached\) are rep-
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 setTo.

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

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

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


Table 1: Experiments.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Depth</th>
<th>Final-Size</th>
<th>Peak-Size</th>
<th>Time</th>
<th>Final-Size</th>
<th>Peak-Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1269a</td>
<td>2</td>
<td>274,845</td>
<td>274,845</td>
<td>73</td>
<td>183,756</td>
<td>183,756</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1423b</td>
<td>10</td>
<td>892,510</td>
<td>966,808</td>
<td>3,186</td>
<td>460,028</td>
<td>1,328,722</td>
<td>1,108</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1512c</td>
<td>1023d</td>
<td>38,117</td>
<td>477,598</td>
<td>1269</td>
<td>47,680</td>
<td>494,478</td>
<td>1057</td>
</tr>
<tr>
<td>s3271c</td>
<td>8</td>
<td>872,205</td>
<td>872,205</td>
<td>2,023</td>
<td>287,476</td>
<td>686,012</td>
<td>615</td>
</tr>
<tr>
<td>s3384c</td>
<td>3</td>
<td>49,951</td>
<td>55,241</td>
<td>37</td>
<td>114,068</td>
<td>114,068</td>
<td>23</td>
</tr>
<tr>
<td>s4863c</td>
<td>4</td>
<td>857,398</td>
<td>857,398</td>
<td>6,560</td>
<td>289,997</td>
<td>289,997</td>
<td>1,927</td>
</tr>
<tr>
<td>s5378c</td>
<td>3</td>
<td>183,406</td>
<td>183,406</td>
<td>586</td>
<td>1,125,997</td>
<td>1,125,051</td>
<td>793</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>565,128</td>
<td>565,128</td>
<td>2,078</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s9234.1</td>
<td>8</td>
<td>76,446</td>
<td>637,829</td>
<td>3,148</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s13207.1</td>
<td>9</td>
<td>602,370</td>
<td>602,370</td>
<td>5,317</td>
<td>784,525</td>
<td>1,772,894</td>
<td>4,093</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>931,975</td>
<td>1,772,894</td>
<td>6,449</td>
</tr>
<tr>
<td>s38584.1</td>
<td>3</td>
<td>1,074,276</td>
<td>1,074,276</td>
<td>1,918</td>
<td>1,146,732</td>
<td>1,146,732</td>
<td>591</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>2,568,209</td>
<td>2,568,209</td>
<td>4,497</td>
</tr>
<tr>
<td>dsipb</td>
<td>15</td>
<td>127,121</td>
<td>171,755</td>
<td>215</td>
<td>49,050</td>
<td>594,333</td>
<td>729</td>
</tr>
<tr>
<td>mm9a</td>
<td>3</td>
<td>19,737</td>
<td>31,570</td>
<td>3</td>
<td>30,068</td>
<td>31,570</td>
<td>2</td>
</tr>
</tbody>
</table>

aLGSynth91
bISCAS89
cADDENDUM93
dtraversal completed