1. ABSTRACT

In this paper, we present an adaptive dynamic variable ordering paradigm which is application-dependent and intended for symbolic model checking applications. The impact of the adaptive variable reordering approach is demonstrated on circuits from Intel’s next-generation micro-processors. On large circuits, in particular, our algorithms make verification tasks that would never end finish successfully in a reasonable amount of time. Our approach, to the best of our knowledge, pioneers in applying successfully ROBDD-independent and application-specific heuristics to the domain of dynamic variable reordering.

1.1 Keywords
Binary decision diagram, variable reordering, symbolic model checking

2. INTRODUCTION

Symbolic model checking based formal verification systems largely depend on the efficient manipulation of boolean functions. One efficient representation that has been shown to be useful over a wide range of functions is Reduced Ordered Binary Decision Diagram (ROBDD) [11,2]. It is well-known that the size of a ROBDD is strongly dependent on the order of the variables in its support. Finding a good variable ordering can make the difference between generating an efficient representation for a function and not being able to represent it at all.

Given a logic function, the problem of finding a variable order which leads to a minimum sized ROBDD for the function has been shown to be NP-hard [5,7]. Practically, the way to find a good order is to construct a good initial order by algorithms that exploit knowledge of the model structure and then to improve the order dynamically [3] during the run of the model checker. Existing state-of-the-art symbolic model checkers like SMV [10], VIS [9], and Rulebase [12] provide variants of Rudell’s dynamic sifting algorithm [6] to solve the dynamic variable ordering problem. Sifting often produces quite good orders, but tends to be very time consuming for large models.

This paper introduces a new paradigm to the usage of variable reordering in symbolic model checking systems. Our approach is based on four major observations on the existing dynamic variable reordering mechanisms present in today’s model checkers:

• The dynamic reordering algorithms greedily look for variable permutations that will result in smaller ROBDD size. The “size above all” philosophy of reordering algorithms in the domain of symbolic model checking contradicts the initial reason for which the reordering is invoked (i.e., to enable the symbolic model checking process to successfully complete in a reasonable time and space). If the reorder time is more than the speed-up obtained because of the reordering, the reordering time spent can not be justified even if the size of the overall ROBDD was reduced at the time of reordering.

• The existing reordering algorithms get invoked only when the number of ROBDD nodes allocated by the application exceeds a predefined threshold that gets updated during the run. Clearly, the reordering process takes a longer time and uses more memory if invoked when the number of ROBDD nodes allocated by the application is considerably large. Forcing reordering at certain critical phases of model checking, when the ROBDD size is below the reordering threshold, may save reordering time and even prevent the application to reach the reordering threshold at a later time.

• The reordering of only a small subset of the variables results in significant savings in the size of the ROBDD at each invocation of reordering. This is mainly observed when the dynamic reordering is invoked on top of a good initial variable order.

1 If several properties on the same model are to be verified; and getting a good order facilitates the verification of the remaining properties, reordering maybe useful even if it takes more time than the speed-up.
The widely used cost function used in dynamic reordering algorithms to find the optimal location for each variable in the ROBDD aims to optimize the size of the overall ROBDD in the system and not the size of particular functions (e.g., transition relation, reachable states) that are of utmost importance during the whole life-time of the verification process.

The above observations reflect the fact that the dynamic reordering algorithms integrated into today’s popular model checkers are application-independent. Actually, variable reordering algorithms are not part of the core algorithms of the model checkers. They belong to the application independent BDD packages used by these systems. Furthermore, Rudell [3] has promoted this characteristic of dynamic variable reordering mechanism as one of its main advantages.

Additionally, we have observed that current dynamic reordering heuristics do not make use of the circuit structure (i.e., the connectivity graph of the variables), which provides valuable information to decide which variables to move and to where to move them. The connectivity information is used only for the initial order generation and ignored afterwards during the reordering.

In this paper, we present a dynamic reordering mechanism that adapts itself dynamically to the needs of the symbolic model checking run. The adaptive variable reordering paradigm takes into account the context of the verification run at the time of invocation to decide which subset of variables to reorder. The reordering heuristics that decide on the optimal positions for these variables also take into account the phase of the verification, and the functions that are being built at the time of invocation. Moreover, we have made use of the circuit structure to decide which variables to reorder at each phase of the reordering. We have proven the concept of adaptive variable ordering in the domain of symbolic model checking run by integrating our algorithms to the SMV based verification core of our formal verification system. The impact of our approach is demonstrated on circuits from Intel’s next-generation micro-processors. On large circuits, in particular, our algorithms make verification tasks that would never end finish successfully in a reasonable amount of time.

This paper is organized as follows. Section 3 summarizes the previous work in this area, and indicates how our approach differs from the prior efforts in the variable reordering domain. Section 4 gives an overview of the adaptive variable reordering paradigm that is tuned for symbolic model checking needs. In Section 5, we report the results on large Intel next-generation circuits using our algorithms. We compare our results with the existing variable reordering mechanism integrated to our formal verification system that is very similar to the dynamic sifting algorithm in [3] and show a significant improvement over the current reordering algorithms. Section 5 also introduces a visual profiling tool [14] that has been used to tune the heuristics of adaptive variable ordering. The last section suggests future improvements using this technique.

3. PREVIOUS WORK

The variable ordering problem has two major aspects: 1) finding a good initial order, and 2) dynamic reordering during the run of the application. In this paper, we mainly address the second aspect of the variable reordering problem, and apply previously used heuristics to get good initial orders in the context of dynamic variable reordering.

The heuristics [1,2,4,7,8] that have been developed for an initial variable order in the domain of symbolic model checking aim to minimize the size of the transition relation. Toutati et al.’s algorithm [2] greedily tries to find an ordering which minimizes the cumulative support of the latches. In [1], Aziz et al. use process communication graph complexity [22] to derive the upper bounds on the size of the ROBDD for a specified ordering. They use the bounds to formulate fast heuristic algorithms for variable ordering tuned to minimize the size of the transition relation. Similar results have been shown for combinational circuits in [10,16]. In [4], Block et al. report on using the communication graph based ROBDD-independent heuristic [7] similar to [1] to find a good initial state variable order. Our work is not innovative in the domain of initial variable ordering. We use a non-greedy fast search mechanism with cost functions similar to [1] to come up with a good initial order. The innovation in our approach is that we use Toutati’s heuristic, which aims to minimize the cumulative support of the latches, in a new domain, namely, for ordering the next state functions that have to be evaluated to generate the monolithic transition relation. Section 4 describes the motivation of applying this heuristic to the domain of next-state function evaluation ordering and its relation to dynamic variable ordering.

To improve variable orders dynamically, state-of-the-art algorithms [9,18,19] are variants of Rudell’s [3] sifting algorithm: group sifting, symmetric sifting, converging sifting. Sifting often produces quite good orders but tends to be very time consuming.

Our approach follows the goal of Meinel & Slobodova [17] to improve the time performance of sifting even if it implies a penalty in the final size of the ROBDD. Meinel & Slobodova speed up the sifting by restricting the reordering of the variables to blocks that are determined by ROBDD-dependent metrics. The adaptive reordering differs from [17] in two major ways. First, we enhance the performance of the sifting in the context of symbolic model checking. We use ROBDD-independent heuristics to estimate which variables need to be repositioned. At each invocation of reordering, instead of trying to find an optimal position for each variable in the ROBDD, we find an optimal position only for the variables that effect the function that is being built at the time of invocation. We do not restrict the reordering to blocks of possible positions for each variable to be reordered as does [17]. The speed-up gained in our approach is mainly in restricting the group of variables to be reordered. We also force reordering at critical stages of model checking to prevent the application from reaching the
reordering threshold or consuming a large number of ROBDD nodes at a later stage.

The experimental results demonstrate that adaptive reordering can give a speed-up both in time and space. Forcing reordering at critical stages of model checking results in more frequent calls to reordering, however, the time spent in reordering is shorter, since not all variables are candidates for reordering and the number of ROBDD nodes allocated by the system at the forced reordering stages is relatively small. Our approach, to the best of our knowledge, pioneers in using successfully ROBDD-independent heuristics in the domain of dynamic variable reordering. The methodology of deciding at each invocation which variables to sift and model checking phase dependent invocation of variable reordering is also unique to our adaptive variable reordering approach.

4. ADAPTIVE VARIABLE REORDERING PARADIGM

4.1 Objective
In the adaptive variable reordering paradigm, our aim is not to find the local minimum for the ROBDD at the time of invocation, but to reduce its size in order to enable the symbolic model checking process to complete successfully with reasonable time and space consumption. We observe that the most critical performance metric to the user of the model checking system is CPU time, as long as memory consumption does not surpass the available resources.

4.2 Overview
The adaptive dynamic reordering approach uses application-specific information:

- The connectivity graph of the functions that are being evaluated.
- The symbolic model checking phase at the time of invocation of reordering.

This information was ignored in the previous applications of dynamic reordering. Additionally, we make use of the context of the symbolic model checking run to decide when to invoke the variable reordering process. Although our approach makes use of application-specific information, this information is passed to the dynamic reordering mechanism in a way that does not violate its application-independent implementation in generic BDD packages.

Moreover, our algorithm adapts the rate of reordering invocation and reordering threshold during the run by taking into consideration the run-time metrics (e.g., CPU time, ROBDD nodes allocated, bytes allocated).

4.3 Features
The novel features of adaptive reordering are: 1. Reordering a selected group of variables at each invocation of reordering. 2. Forcing reordering at critical stages of model checking. 3. Taking into consideration the size of a subset of important functions in addition to overall ROBDD size when looking for an optimal position for a variable. 4. Adapting the reordering parameters by taking into account the run-time metrics.

In this section, we concentrate on 1 and 2. Section 4.3.1 describes the heuristics to choose which variables to reposition at each invocation of the dynamic reordering. This section also describes the order in which the next state functions are evaluated to facilitate more effective selection. In the results section (Section 5), we give insight on the impact of these features.

4.3.1 Heuristics to choose the variables to be sifted
The main idea is to reorder only the variables affecting the function that is being built at the time of invocation. The definition of the function that is being built changes during the run of the model checker. For example, during the monolithic1 transition relation build, the functions to be built are next state functions of the sequential devices (i.e., latches/flip-flops) of the model under test. If the reordering is invoked during the build of the next state function, only the variables in the support of this function are reordered.

To increase the locality of the variables being reordered, adaptive reordering computes an evaluation order for the next state functions that minimizes the cumulative support of the sequential elements in the model being verified. We use a non-greedy, fast search algorithm that minimizes the following cost function for the variable permutation α:

\[
\text{cost}(\alpha) = \sum_{1 \leq j \leq n} | \bigcup_{1 \leq i \leq j} \text{supp}(f_{\alpha i}) |
\]

where |A| denotes the cardinality of A, \( f_{\alpha i} \) denotes the next-state function of the latch \( \alpha_i \) and \( \text{supp}(f_{\alpha i}) \) corresponds to the set of variables in the support of \( f_{\alpha i} \), and \( n \) is the number of sequential elements.

The use of this heuristics in the domain of dynamic reordering during the model checking is as follows:

- Find an initial order for all the variables (i.e., input and state variables) using heuristics similar to [1].
- Find an order \( \text{ord} \) for the state variables that minimizes the cumulative support when the state variables are interleaved with the next state variables. Evaluate the next state functions in the order depicted by \( \text{ord} \) in the transition relation build phase of the model checking.
- Record which functions of importance are being evaluated during the run of model checking.

---

1 If partitioned transition relation is being built, the variables in the support of the functions that make up the partitions will be candidates to be reordered.
If the reordering is called (i.e., either invoked automatically because the reordering threshold has been reached or forced explicitly) during the evaluation of a function of importance (e.g., the next state function of a latch), reorder only the variables in the support of this function.

Note that the ordering of next state function evaluation strengthens the efficiency of the heuristics used to select which variables to reorder at each invocation of the ordering during the transition relation build. The evaluation order results in variable reordering first to minimize next state functions with common logic and later the ones with the disjoint logic. Therefore, it reduces the number of variables that affect the overall size of the ROBDD at each invocation of the reordering. We have found that this technique is especially effective when trying to verify large designs when the verification engine’s capacity is challenged during the transition relation build.

4.3.2 Forcing the reordering

We have observed that forcing variable reordering at certain phases of the model checking, independent of the reordering threshold, may dramatically improve the model checking performance. It is clear that reordering takes more time if the number of ROBDD nodes allocated is large at the time of invocation. Therefore, forcing reordering at critical times during model checking before reaching the reordering threshold can even prevent the application reach the threshold. In Section 5, we illustrate examples that justify this point.

The adaptive variable reordering has a few modes of operation that affect the rate of forced reordering. When dealing with test cases challenging the capacity limits of the verification task, variable reordering is forced more frequently (e.g., after every next-state function evaluation, after every fixed-point iteration). The intuition here is to optimize the ordering of variables before reaching reordering thresholds or the allocation of a large number of ROBDD nodes by the application. The reordering time can be very large when invoked on ROBDDs over 1M nodes. The chances of running out of memory during the reconstruction of ROBDD in search for an ordering that produces a smaller representation is higher when working with large ROBDDs. For smaller test cases, the forcing is done only after major model checking phases (e.g. transition relation build). The modes of operation are currently controlled by the user. However, we plan to automate the choice of the modes internally based on the run-time information.

4.3.3 Application-specific information with application independent implementation

Although the objective of having an application-independent variable reordering mechanism is of utmost importance, we believe a big price is paid when the application-specific information is ignored during the variable reordering. Furthermore, we believe application-specific information can be passed to the dynamic reordering mechanism in a way that does not suppress its application-independent characteristic. Therefore, in our approach, the usage of variable reordering is application-specific, while the implementation stays application-independent.

5. EXPERIMENTAL RESULTS

In this section, we present some results comparing the adaptive dynamic reordering approach to conventional dynamic sifting. As mentioned before, adaptive dynamic reordering is integrated into the SMV based model checking engine of our formal verification system [13]. The version of the tool with the new approach is compared against the version with the conventional dynamic sifting reordering mechanism similar to the one implemented in [3].

For each of the methods, we verify properties on real designs when the same initial order is provided to the verification engine. Each verification task consists of the verification of multiple properties, and the same input and SMV configuration was provided for both methods. The verification time reported does not take into account the time spent to generate the initial order. Run times were measured on an IBM RS6000 machine with 512M memory.

The six test cases reported in Table 1. are real verification examples from Intel’s next-generation processors. The test cases are mainly control dominant, although real1, and real2 are memory-intensive circuits. The same initial reordering threshold is used for the conventional and adaptive sifting algorithms.

The real1, real2, and real4 test cases, as can be judged from the data in Table 1, challenge the capacity limits of the verification engine. Most of the time in real1 and real2 is spent in transition relation build. The adaptive reordering succeeds in building the transition relation and completes the verification, whereas the conventional reordering fails to build the transition relation. The overall memory consumption of real1 is six times less than the conventional mechanism at time-out. The real4 test case takes a long time during the transition relation build and the specification evaluation stage. Verification with the conventional reordering mechanism times out at specification evaluation stage, although succeeds in building the transition relation. The real3 test case is an excellent example that demonstrates the benefit of forcing reordering at critical times during model checking. Although the adaptive reordering mechanism invokes reordering nine times more than the conventional mechanism, the reordering time is four times less, and the final transition relation size is twenty times smaller.

5.1 Visualizer to Tune Variable Reordering

We have used a visual profiling tool [14], PALETTE, to benchmark the adaptive dynamic reordering paradigm against the conventional dynamic reordering mechanism. Furthermore, the tool has helped us tune the heuristics used by the adaptive approach. PALETTE visually displays model checking phases (e.g. reachable states build, transition relation build, or dynamic reordering) against
selected profiling metrics (e.g. BDD nodes allocated). As mentioned above, the adaptive variable reordering approach takes into account the model checking phase to decide when to force the reordering and which variables to reposition. Therefore, a tool like PALETTE that enables symbolic model checking with phase-wise profiling was very helpful in the development of the new reordering paradigm.

Figure 1 demonstrates the displays produced by the tool for the real3 testcase. The PALETTE displays for this test case give insightful information on the importance of forced reordering and reordering a selected subset of variables. As can be seen from the benchmarking data on real3 provided in Table 1, dynamic reordering is invoked twice using the conventional reordering mechanism. The second reordering takes most of the time, and the memory consumption during this stage goes from 35M to 56M (See third view from the top in Figure 1). The double memory consumption is due to the reconstruction of the ROBDD during the variable reordering phase. On the other hand, using the adaptive reordering approach (See the topmost two views in Figure 1), the second reordering is invoked at about the same threshold and lasts a short time and the coming reorderings do not take a long time and improve significantly the transition relation size. Furthermore, the memory consumption stays the same.

6. CONCLUSIONS AND FUTURE WORK

We have shown that application-dependent information can enhance the use of dynamic reordering by integrating adaptive variable reordering approach into our symbolic model checking engine. Furthermore, we have demonstrated that the use of application-dependent heuristics does not necessarily make the implementation of the dynamic reordering mechanism application-specific. The impact of our approach is demonstrated on circuits from Intel’s next generation micro-processors. In particular, on large circuits our algorithms make verification tasks that would never end finish successfully in a reasonable amount of time and space.

The adaptive reordering paradigm to the best of our knowledge pioneers in applying ROBDD-independent heuristics to the domain of dynamic variable reordering. The selection of the variables to be reordered at each invocation of the reordering and forcing of reordering at critical phases of the model checking is also unique to our paradigm.

Considerable work remains to be done. The heuristics to estimate which variables to reorder at each invocation have to be tuned by trying our approach on more real-life examples. Additionally, we would like to explore the possibility of using ROBDD-independent heuristics to decide where to reposition the variables. The use of heuristics that do not require the reconstruction of the ROBDDs will speed-up the reordering process a great deal. Furthermore, we plan to integrate learning techniques to the estimation of which variables to reorder, and to the adaptation and tuning of the reordering parameters.

7. ACKNOWLEDGEMENTS

The authors would like to thank Guy Gilboa, Barukh Ziv, Sela Mador-Haim and Doron Stein for their comments and suggestions in improving this work.

8. REFERENCES


<table>
<thead>
<tr>
<th>Testcase</th>
<th>Variables (Seq.elems/Inputs)</th>
<th>CPU time (in units of seconds)</th>
<th>Bytes Allocated</th>
<th>Transition size (in units of BDD nodes)</th>
<th>Reorder time/Number of reorderings</th>
</tr>
</thead>
<tbody>
<tr>
<td>real1</td>
<td>273 (264/9)</td>
<td>9912</td>
<td>30M</td>
<td>30684</td>
<td>9273 / 122</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time-out</td>
<td>173M</td>
<td>time-out</td>
<td>2733 / 4 time-out</td>
</tr>
<tr>
<td>real2</td>
<td>269(260/9)</td>
<td>4223</td>
<td>16M</td>
<td>73246</td>
<td>1864/113</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time-out</td>
<td>173M</td>
<td>time-out</td>
<td>time-out</td>
</tr>
<tr>
<td>real3</td>
<td>99(60/39)</td>
<td>340</td>
<td>32M</td>
<td>7025</td>
<td>281 / 17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1257</td>
<td>57M</td>
<td>196213</td>
<td>1097 / 2</td>
</tr>
<tr>
<td>real4</td>
<td>136(96/40)</td>
<td>64952</td>
<td>124M</td>
<td>350294</td>
<td>34853/481</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time-out</td>
<td>92M</td>
<td>439404</td>
<td>time-out</td>
</tr>
<tr>
<td>real5</td>
<td>78(43/35)</td>
<td>5746</td>
<td>51M</td>
<td>437888</td>
<td>3290/197</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6019</td>
<td>51M</td>
<td>438776</td>
<td>5727 / 4</td>
</tr>
<tr>
<td>real6</td>
<td>101(77/24)</td>
<td>2811</td>
<td>25M</td>
<td>68247</td>
<td>2452 / 102</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4809</td>
<td>25M</td>
<td>134950</td>
<td>1349 / 1</td>
</tr>
</tbody>
</table>

Table 1. The boldfaced data represents runs with adaptive dynamic reordering; whereas the plain data represents runs with the conventional reordering. A 24-hour time-out has been set on the runs. The first column represents the number of variables in the verification task. The second column represents the elapsed user time obtained from the UNIX time command. The bytes allocated by the verification process and the size of the transition relation at the end of the run are represented by the second and third columns. The right-most column represents the CPU-time spent in dynamic reordering and the number of times dynamic reordering is invoked.
Figure 1. The PALETTE screens below visualize the verification run for the benchmark real3 from Table 1, against the profiling metrics “Bytes Allocated” and “BDD Nodes Allocated” respectively. The x axis represents the CPU time in secs. The top two screens correspond to the run with adaptive reordering with respect to BDD and memory byte consumption, and the other two screens represent the profiling information for the conventional dynamic reordering. The colors correspond to different stages of model checking, when the dominant light (i.e., beige in colored print-out) color corresponds to “dynamic reordering” stage. The numbers at the peaks correspond to the maximum y value when the y axis corresponds to the selected profiling metrics (e.g., in the second screen below, the maximum BDD nodes allocated is 1824078). The numbers at the left and right of the displays represent the start and end time of the verification process.